A Study of the Functional Memory Type Parallel Processor(Dissertation_全文)

AUTHOR(S):

Kobayashi, Kazutoshi

A Study of the Functional Memory Type Parallel Processor

http://www.tamaru.kuee.kyoto-u.ac.jp/fmpp/

Kazutoshi Kobayashi

Kyoto University

September 1998

Abstract

This paper describes a memory-based SIMD shared-bus parallel processor architecture, which is called “Functional Memory Type Parallel Processor” abbreviated as FMPP. The FMPP architecture integrates memory and an ALU closely on a single die. All the PEs are connected with a shared bus and laid out in a two-dimensional array like memory and perform the same instructions according to the SIMD manner. The FMPP architecture enables massively parallel computing inside a memory. It has a capability to break the Von Neumann bottleneck where the system performance is limited by the bus performance between memory and CPU.

We have developed four LSIs based on the bit-parallel block-parallel architecture, where a PE consists of several words and a bit-parallel ALU. The first LSI called the BPBP-FMPP with 8 PEs is designed and fabricated for general purpose. A PE consists of 32bit CAM words and a 32bit ALU for numerical and logical operations. The following three LSIs called FMPP-VQ are for a special purpose: vector quantization (VQ). A PE consists of 16 words of 8bit SRAMs and a 12bit ALU. The FMPP-VQ accelerates the nearest neighbor search where the vector nearest to an input is extracted among large number of code vectors. The FMPP-VQ4 with 4 PEs is an evaluation LSI to confirm functionalities. The second FMPP-VQ64 integrates 64 PEs. It performs over 50,000 nearest neighbor searches per second, while its power consumption is 20mW. It can be used for real-time low-rate video compression. The third FMPP-VQ64M is designed for more powerful and low-power computation. Its performance becomes almost twice, while its power consumption is half compared with the FMPP-VQ64. We have also developed a low-rate video compression system using the FMPP-VQ. The proposed multi-stage hierarchical vector quantization algorithm can transmit 10 QCIF frames per second through a 29.2kbps mobile channel.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This paper is a summary of a memory-based SIMD (single instruction multiple data stream) shared-bus parallel processor architecture, which is called the “Functional Memory Type Parallel Processor” abbreviated as FMPP. Almost all the current computing systems are based on the Von Neumann architecture, where a CPU and memory devices are connected with a shared bus. All the data and programs should be transfered between the CPU and memory through the bus. Although the performance of the CPU is rapidly improving, the performance of the Von Neumann system is limited by the performance of the bus or the memory, which phenomenon is called “Von Neumann Bottleneck.” This is because the performance of the memory is not improved faster than that of the CPU and the width of the bus is limited to be narrow. The bottleneck must be alleviated to perform processing inside a memory device. The memory device implies parallel computation capability, since it consists of a two-dimensional array of memory words. All the words can work in parallel. The two-dimensional regular array structure achieves highly dense layout improving four times every three years. The FMPP architecture allows parallel processing inside memory devices. A processing element (PE) consists of some amount of memory cells and an ALU. All the PEs connected with a shared-bus work in parallel according to a single instruction provided through a central control unit (the SIMD control method). The FMPP is suitable for operations where communication between processors or external devices is not so frequent and the same operations are done for huge number of data set. The processing capability is defined by the processor granuality. Fine granuality makes the functionality poor. Coarse granuality enlarges the area. A bit-parallel block-parallel (BPBP) structure is proposed in this paper for middle-grain modest-functional processing. The PE consists of several words and a bit-parallel ALU. Four LSIs have been developed and fabricated based on the BPBP structure. The first emerged LSI is called BPBP-FMPP, which contains eight 32bit PEs and can be applied for general purpose. The following three LSIs called the FMPP-VQ are for a special purpose: vector quantization (VQ). A PE consists of 16 eight-bit SRAMs and a 12bit ALU. The FMPP-VQ can search the vector (pattern) nearest to an input among all the vectors stored in PEs. It is successfully applied to real-time low-rate video compression by VQ. The first FMPP-VQ LSI

contains four PEs to evaluate its functionalities. The second and third attempts integrate 64 PEs to be applied for real-time low-rate image compression. We have also developed an algorithm and a real-time low-rate compression system using the FMPP-VQ.

Chapter 2 gives overview of the functional memory. The functional memory is a memory device with some functionalities. The FMPP can be categorized to the functional memory. The CAM and associative processor architectures are also discussed. Chapter 3 explains the FMPP architecture in detail. The BPBP structure is compared with the other two structures, bit-parallel word-parallel and bit-serial word-parallel. The performance efficiency on the FMPP-based computing system is also argued. The BPBP-FMPP is described in Chapter 4. Chapter 5 introduces the FMPP-VQ architecture, the three LSI implementations and the real-time low-rate video compression system. Chapter 6 summarizes this paper.

Chapter 2

Overview of Parallel Processor and Functional Memory

This chapter describes the overview of parallel processor and functional memory. The functional memory type parallel processor (FMPP) is a parallel processor architecture based on functional memory. First, we address the Von Neumann bottleneck eliminating the performance of the current computing system. Several parallel processor architectures are introduced to break the bottleneck. Then, functional memory and associative processors are described in detail.

2.1 Parallel Processor Architectures to Break the Von Neumann Bottleneck

The current Von Neumann computer architecture confronts the bottleneck where the system performance is limited by the bus performance. This section gives the brief description of the bottleneck and shows several parallel processor architectures to break the bottleneck.

2.1.1 Von Neumann Computer Architecture

In the Von Neumann architecture, computers are composed of a central processing unit (CPU) and a memory unit (Figure 2.1). CPU performs operations according to codes (programs) and data in the memory unit. They are connected with a bus. At the rise time of computers, thousands of relay switches and vacuum tubes form a computer unit. As the emergence of semiconductor devices, the memory and CPU are replaced with discrete transistors. Now they are integrated on LSIs (Large Scale Integrations or Large Scale Integrated circuits). The most popular commercial processor Pentium integrates over 1 million transistors and its internal clock speed becomes over 300MHz. The largest commercial DRAM (Dynamic Random Access Memory) has 256 million bits on a single LSI. In the Von Neumann architecture, all data and codes have to be passed from the main memory to the CPU through the bus. The bandwidth between them is narrow, since the number of pins pulled outside LSIs are limited. Pentium has only 64-bit bus. The access time of DRAM is about 50ns

(20MHz). In these conditions, Pentium can perform 300MIPS (Mega instructions per second), but the DRAM gives only 160M bytes of data and codes per second to the CPU. The bus degrades the performance of the CPU, which is so-called "Von Neumann bottleneck." To compensate the Von Neumann bottleneck, all of current commercial CPUs have memory hierarchy as shown in Figure 2.1. The memory hierarchy virtually shortens the access time of the DRAM if the accessed data exists on cache memory (Cache Hit). The actual access time, however, becomes longer because the actual distance from the CPU to the DRAM becomes longer. That produces severe problems in some applications. For example, these hierarchical structure of cache memory is not so effective for image processing. Image processing usually applies the same operations to image data. The image data of video sequence amounts to huge size. In the JPEG, the famous DCT-based image compression algorithm, the image data is divided into a block, each of which includes 8×8 pixels. Each block has no relation with others. In this situation, the first cache in the CPU can store a single block data, which accelerates processing. But the second cache between DRAM and CPU does not contribute the processing speed. It merely prolongs the access time of DRAM.

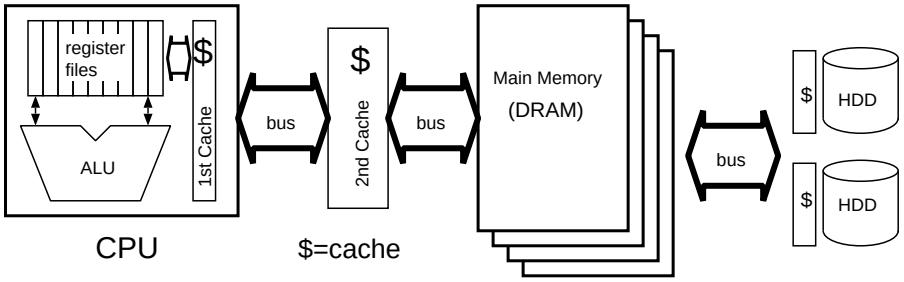


Figure 2.1: Von Neumann computer and its hierarchical memory structure.

As for the power dissipation, an off-chip bus to connect the CPU and memory dissipates a large amount of power. Reference [Wat98] mentions that an external pin-to-pin I/O connection yields 50pF of stray capacitance, while an internal I/O connection yields only 1pF which is 50 times smaller. The dissipated power is proportional to the value of stray capacitance. If we implement the CPU and memory in a single LSI, the power dissipation must be minimized. Recently, such a challenge called "Merging memory (DRAM) and Logic" becomes very popular. The deep sub-micron process on the current VLSI technology actualizes a mixture of logic and DRAM on a single LSI. Some LSI vendors develop commercial products implementing some amount of DRAM and logics on a single die[INK+95, WFY+97]. eRAM^{TM}by Mitsubishi Electric Corp. stands for "embedded random access memory"[ERA]. The 3D-RAM[INK+95] is one of LSIs of the eRAM architecture. It integrates Z-compare or α-blend units to be applied to 3D-graphic applications. A single LSI contains 10Mbit DRAM and an SRAM cache with a single ALU for Z-compare or α-blend. Several

LSIs simultaneously work to complete 3D-graphic applications.

2.1.2 SIMD Parallel Processor to Solve the Von Neumann Bottleneck

Several approaches can be taken to break the Von Neumann bottleneck. One approach is to have multiple CPUs work in parallel, which is called "parallel processors." Pentium now integrates SIMD processors, which is called MMX extension, which is suitable for image processing or video game. Parallel processor is a key technology to obtain more powerful and effective computation on LSIs. Parallel processor is categorized in two by its memory architecture: distributed memory and shared memory. In the distributed memory architecture, a processor has its own memory, while all the processors shares common memory in the shared memory architecture. The distributed alleviates the bottleneck more than the shared, since the bandwidth between memory and processor is extended according to the number of processing elements (PEs). A complex control method usually makes it difficult to describe a parallel program and makes the area of the PE larger. General parallel processors can also be grouped into two major categories by the control method. One is SIMD that means "Single Instruction Multiple Data Stream." The other is MIMD that is an abbreviation of "Multiple Instruction Multiple Data Stream." On the SIMD, all processors work simultaneously according to the same instruction. On the MIMD, each processor performs its own instruction. An SIMD parallel processor can be implemented in a smaller area than an MIMD parallel processor, since a PE of the MIMD should have its own control logic. All PEs of the SIMD, however, can be controlled by a common control logic. The number of PEs on a single die should become larger in the SIMD architecture. Thus, the SIMD distributed-memory parallel processor is a good candidate to break the bottleneck.

We should consider some more parameters to implement SIMD distributed-memory parallel processors. Here, these three parameters are chosen to categorize them : processor granuality, processor functionality and communication network. They have strong correlation with each other. Fine processor granuality usually makes the functionality of a PE poorer. Complex communication network always makes the area of an LSI larger. We introduce several implementations of SIMD distributed-memory parallel processors by those three parameters. Table 2.1 shows the three implementations of the SIMD distributed-memory parallel processors.

Connection Machine[Hil87] is an SIMD parallel processor. In the first system called CM1, each PE consists of 4kbit memory and a bit-serial ALU, which is very simple. The CM1 consists of 64k PEs connected with a complex flexible network called "hyper-cube." A software programmer can design a network of processors as he want. Connection Machine is developed for general purpose.

Content Addressable Memory (CAM) is a memory device which can associate address from contents of memory. Detail descriptions are shown later in Section 2.2.1 and Section 3.3.1. It is

Table 2.1: SIMD distributed-memory parallel processor implementations.

	Granuality	Functionality	Network (Type)
Connection Machine	fine	poor	Complex (Hyper-cube)
CAM	very fine	very poor	Simple (Bus-connected)
IMAP	coarse	medium	Simple (Bus-connected)

usually regarded as memory than parallel processors. But it can be applied to parallel processing using its associative capability as shown in Section 3.3.1.

IMAP stands for Integrated Memory Array Processor proposed by a group of NEC[FYO92]. It is an SIMD parallel processor architecture merging SRAM and logic. The IMAP LSI[KNA+95] integrates a 2MB SRAM with 64 PEs. The block diagram is shown in Figure 2.2. A 64kb SRAM macro is assigned to two PEs. They can directly communicate with the assigned SRAM macro.[1] Each PE consists of several 8bit registers and an 8bit ALU. Its peak performance becomes 3.84 GIPS, but its power consumption is 4W. An image processing system connected to the PCI bus is already commercially available[NEC].

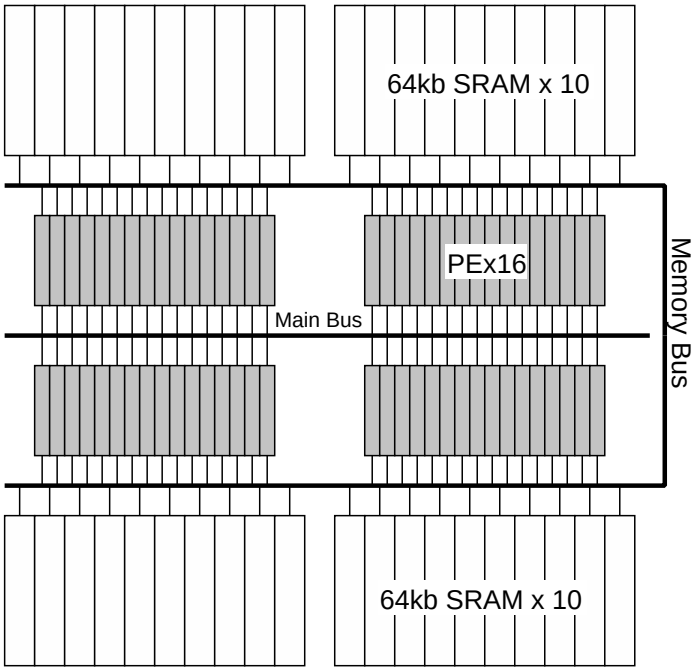


Figure 2.2: IMAP LSI.

As described above, various distributed-memory SIMD parallel processor architectures are available. In the Connection Machine architecture, complex network reduces the integration density. The

[1] 8 SRAM macros are redundant.

progress of the current VLSI technology enlarges the integration density year by year. The memory device enjoys the progress enormously, since its two-dimensional array structure and shared-bus simple network are very much suited to the VLSI technology. Thus, the parallel processor architecture based on the memory structure may hugely enjoy the VLSI technology. The CAM has the capability to perform parallel processing inside memory. But its functionality is too poor. The IMAP integrates multiple processor and memory devices on a single die. But memory and processor are separately designed and the processor granuality is relatively coarse.

In this paper, we focus on a memory-based SIMD shared-bus parallel processor architecture called FMPP. FMPP stands for Functional Memory type Parallel Processor. The FMPP architecture enables fine-grain highly-functional parallel processing inside memory. It allows numerical operations inside memory. In the next section, we introduce functional memory and associative processor before describing the FMPP architecture.

2.2 Functional Memory and Associative Processor

Functional memory can be described as a memory including some simple functions such as content addressing. It is proposed by Kohonen[Koh87] as "associative memory." The original associative memory is some kind of conceptual one. It can associate a target value from several key values like our brain. Kohonen implemented an optical associative memory to retrieve a full-sized image from an incomplete image. On the other hand, content addressable memory (CAM) is an actual LSI implementation of associative memories. Conventional memories like DRAMs or SRAMs associate data from an address, while the CAM associates an address from data. Associative processor is a parallel processor architecture to perform processing using its associativity. Here we explain the CAM and the associative processor in detail. Implementations of the FMPP proposed in this paper are based on the CAM architecture.

2.2.1 Content Addressable Memory

Ogura et al. implemented a 4kb content addressable memory[OYN85] in early 80's as shown in Figure 2.3. Its memory cell is shown in Figure 2.4 along with a memory cell of a conventional 6-transistor SRAM. The two pass transistors denoted by dashed circles work as a pass-transistored XNOR (exclusive-nor) logic. Let the CAM cell store A and supply $\overline{B}$ and B to the two bit lines b0 and b1 respectively. Note that the supplied value is inversed: b0$=\overline{B}$ and b1$=B$. The output node C becomes logic high if $(A = 1)\&(B = 0)$ or $(A = 0)\&(B = 1)$. It means $A \oplus B$ (XOR). To obtain words matched to a key value, the multiple bits of a CAM cell form a single match line that works as a wired-NOR of all results from the XOR gates (See Figure 2.5). In the initial condition, the Match

line is precharged. A key value is supplied to the CAM word, then the $\overline{\mathsf{PRE}}$ is activated. If $A = B$, the match line keeps logic high, or it is discharged since an XNOR gate where $A[i] \neq B[i]$ becomes true. The search flag connected to the Match line stores the search result. We call the operation "search operation." The search operation may have multiple search flags become true. The multiple response resolver resolves the lowest address among CAM words search flags of which are true. The signal E_R becomes false if there is no true search flag. The garbage flag invalidates the search result. It is usually used to read all the addresses search flags of which are true. When the lowest word (the word address of which is the lowest of all) is read out, its garbage flag becomes true. Then, the multiple response resolver produces the second-lowest address. Such an operation is called "Multiple Response Resolution." The CAM also has a functionality of parallel write operation: writing all the words whose search flags are true.

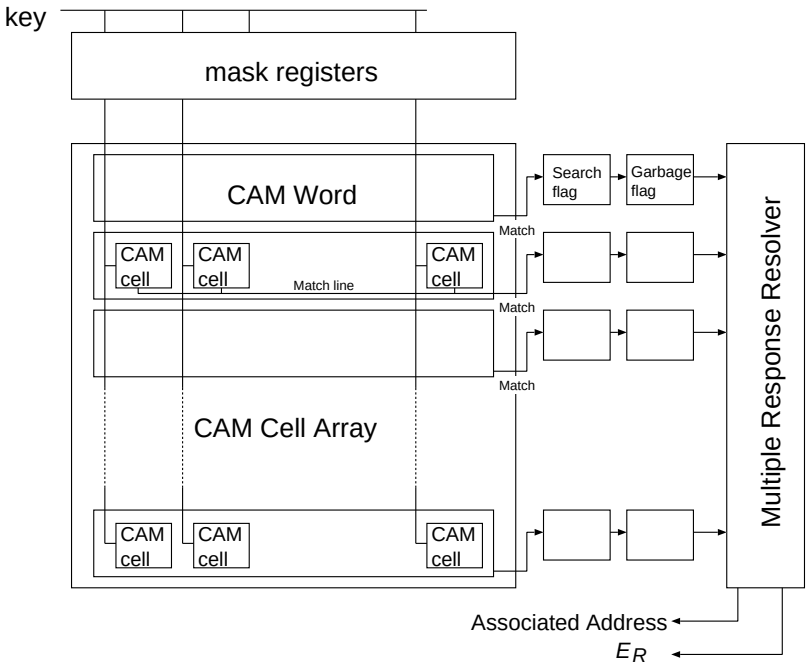


Figure 2.3: Block diagram of a 4kb CAM.

The mask register located at the top of the CAM cell array in Figure 2.3 masks specified bits. The two bit line b0 and b1 become logic low at the masked bit. Thus, the XNOR gate of the bit always generates the false output regardless of the bit data. Thus, the specified bit is masked on the search operation. The mask signal in the CAM cell is also connected to the mask register. It is used to prohibit the write operation to the specified bit.

They applied the CAM to Prolog machines[NO90]. The CAM accelerates the back-track scheme. The garbage flags support the garbage collection in Prolog. They have been developing high-density CAM LSI implementations[FOT93, ONB+96, ON97]. The latest CAM LSI in [ON97] integrates

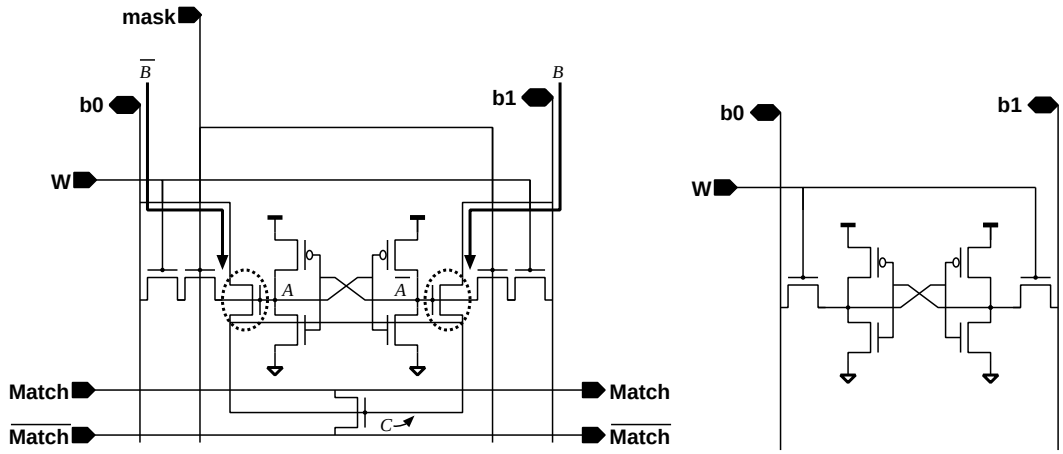


Figure 2.4: Memory cells of a CAM(left) and a conventional 6-transistor SRAM(right).

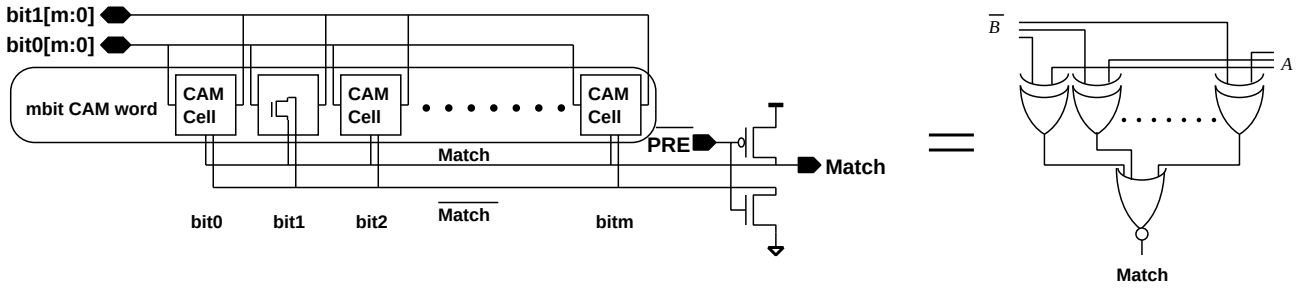


Figure 2.5: Match line on a CAM cell.

366k-bit on a 16.5×16.5 mm^2 die, which is applied to image processing.

2.2.2 Associative Processors

S.S. Yau and H.S. Fung surveyed associative processors in Reference [YF77]. An associative processor can generally be described as a processor which has the following two properties:

1. Stored data items can be retrieved using their content or part of their content (it is called content addressing).

2. Data transformation operations both arithmetic and logical can be performed over many sets of arguments with a single instruction (it is called parallel computation).

Although, the CAM has only the former property, we can regard the CAM as an associative processor. Almost all implementations to be categorized into associative processors are based on the content addressing capability of the CAM. In the rise time of associative processors, they were applied to various fields, for example, geometrical problems[SKO90], a database accelerator[WS89],

a Prolog engine[NO90] and etc. But current target applications tend to image processing. This may be because advantages obtained by these embedded associative processors are soon supposed by commercial micro processors which have remarkably been improving. In the area of image processing, however, these associative processors get a great advantage over the micro processors on the processing speed and power dissipation.

The architecture of associative processors can generally be classified into three categories according to the processor granuality. The three categories are bit-oriented, word-oriented, and block-oriented associative processors. The bit-oriented associative processors is the most fine grain one, which PE consists of a single-bit memory cell with an ALU. A single word with an ALU forms the PE of the word-oriented associative processors. The block-oriented associative processors are implemented as the bit-parallel block-parallel FMPP in this paper. It consists of several words of memory cells and an ALU. Comparison of these three architectures is discussed in Section 3.2. The word-oriented architecture is the most widely-spread and famous, since it can easily be implemented to add a specific word-oriented ALU to a CAM word as shown in Figure 2.6. The ALU retrieves the single bit data through the match line of a CAM word.

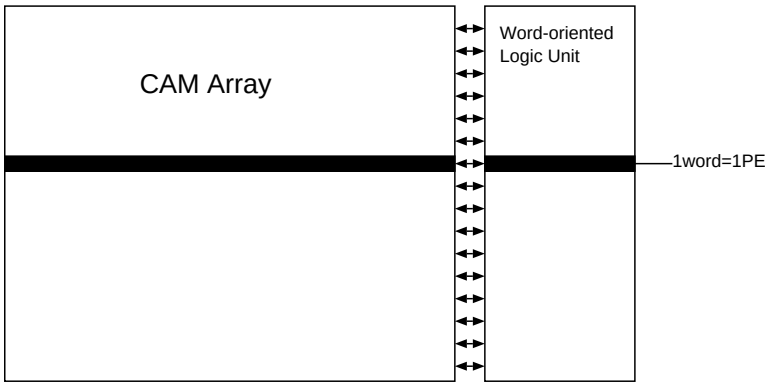


Figure 2.6: General bit-serial associative processors.

2.2.3 Implementations of Associative Processors

Here, several implementations of associative processors are introduced.

A group of Waseda University[Was] proposed a CAM-based hardware engine for geometrical problems[KNK+92]. They developed a 4kbit CAM to accelerate threshold search, extremum search and parallel numerical operations. Numerical operations are done in bit-serial in an ALU attached to a word.

A group of Tohoku University[Toh] develops a multiple-valued CAM [HAK97]. A cell of the CAM is a floating-gate MOS transistor similar to EEPROM cells (Figure 2.7). The floating-gate MOS

transistor stores 4 states by controlling the threshold voltage. They just propose a circuit diagram of the CAM. They will apply it to fully parallel template-matching operations. They carry out another research of intelligent vehicles, where a ROM-type CAM is applied to collision avoidance[HK96].

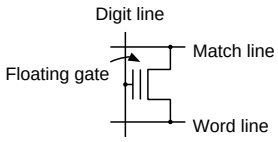


Figure 2.7: Multiple-valued CAM cell.

A dynamic associative memory processor has been proposed by Sodini et. al. in [HS92]. As shown in Figure 2.8 a two-dimensional network connects all the PEs. A PE consists of associative parallel processors which can be word-oriented or fully-parallel. A memory cell is called a dynamic content-addressable parallel processor cell as shown in Figure 2.9. Image processing such as smoothing is introduced as an effective application on the dynamic associative memory processor, with each PE assigned to a single pixel. They fabricated a 256-element associative parallel processor LSI[HS95]. Currently, they have proposed and fabricated a pixel-parallel image processor based on the DRAM-merged logic architecture[GS97]. A PE has the similar structure in the dynamic associative memory processor. But a memory cell is replaced with a conventional DRAM cell. It integrates 128×128 processors on a 78.6mm^2 die.

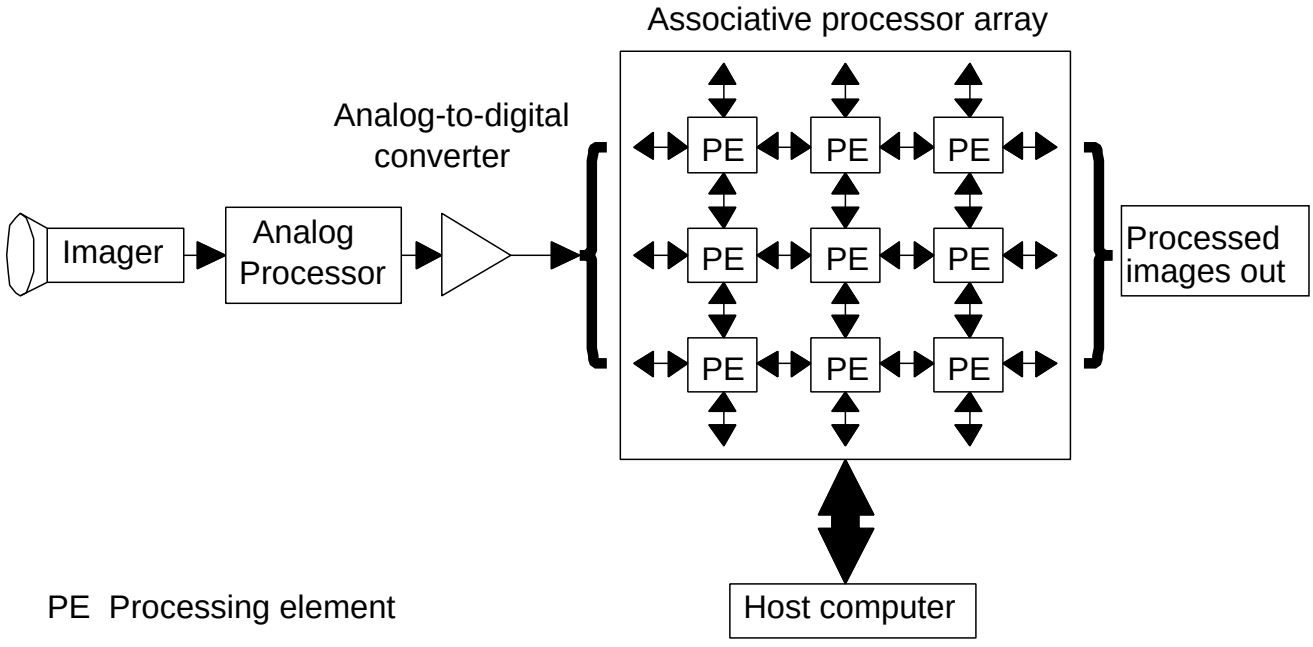


Figure 2.8: A pixel parallel image associative processor[HS92, GS97].

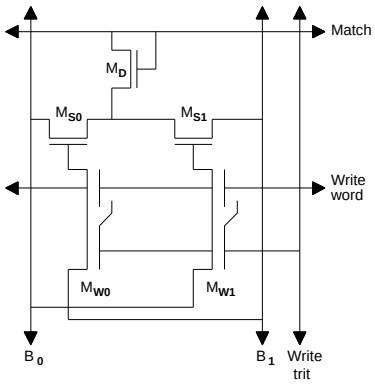


Figure 2.9: Dynamic content addressable memory cell.

Computational RAM (C∗RAM) is a memory-SIMD hybrid architecture where each column of memory has an associated processing element[ESS92]. Figure 2.10 shows an implementation of C∗RAM. There are 64 bit-serial PEs. A 1kbit memory column is assigned to each PE. It is applied to several image processing application including vector quantization. The detail description of applying vector quantization is explained in Section 5.7 compared with the implementation of the FMPP.

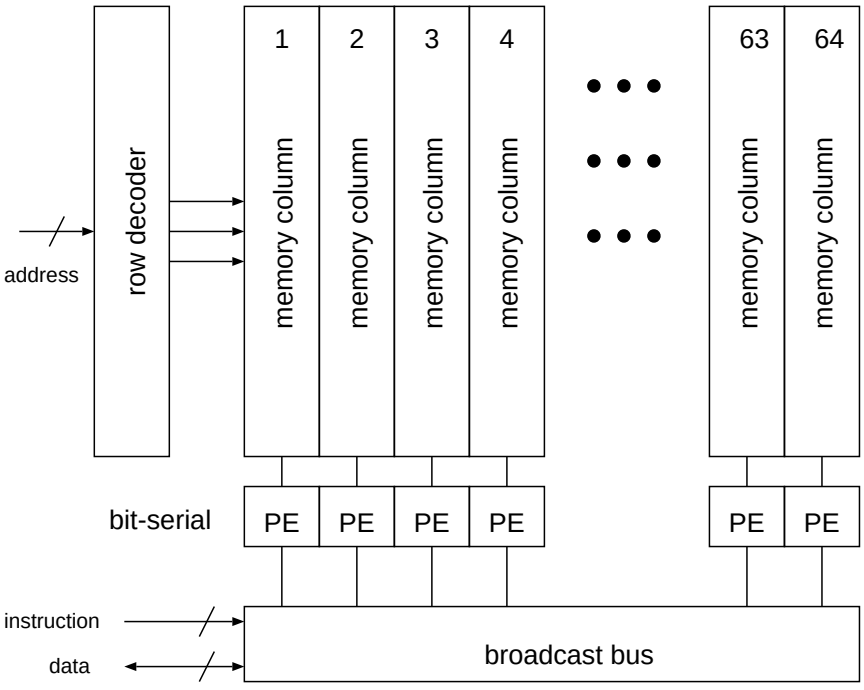


Figure 2.10: An implementation of computational RAM.

2.3 Summary of the Chapter

Here, parallel processor and functional memory are briefly discussed. Von Neumann architecture has been used in the current computer system. At the emergence of the computer, the CPU and the memory are separately fabricated. But now both can be integrated on a single die. The challenge to merge memory devices and processors on a single LSI has just started recently owing to the current rapid progress of integration density. It extends the bandwidth between memory and processors considerably. But the speed gap between DRAMs and processors still remains. The gap should be compensated by memory hierarchical structure. But it prolongs the bus length. If some amount of processing can be done on memory devices, the system performance will be promoted. The functional memory architecture attaches simple processing capability to memory devices to enable on-memory processing. The CAM, the most famous widely-used functional memory can detect an address from its content. It can be regarded as a parallel processor where each word becomes a processor. Its two-dimensional structure is very much suited to the current VLSI technology. The functional memory type parallel processor, FMPP architecture described in the next chapter can be categorized to the functional memory. It is a memory-based SIMD shared-bus parallel processor. The FMPP integrates fine-grain memory-based PEs in a two-dimensional array. The features of SIMD and shared bus enhances the integration density. Huge number of processors on a single LSI perform massively parallel computing.

Chapter 3

Functional Memory Type Parallel Processor: FMPP

In this chapter, we introduce the Functional Memory Type Parallel Processor (FMPP) architecture in detail. Three structures are available for the FMPP architecture: fully-parallel (bit-parallel word-parallel), word-oriented (bit-serial word-parallel) and block-oriented (bit-parallel block-parallel). This paper focuses on the block-oriented implementations. Finally, we compare the performance efficiency between a conventional Von Neumann computer and an FMPP-based computer.

3.1 Features of the FMPP

The FMPP is a memory-based SIMD share-bus parallel processor which can enjoy some direct benefit from memory VLSI technology. The FMPP architecture is schematized in Figure 3.1.

The features of FMPP are summarized as follows[YWST91, Yas91].

Memory-Based Simple Structure. The FMPP has a memory-based simple two-dimensional array structure like an LSI memory. Each processor contains a bit, a word, or a group of words. We can obtain a very large parallel computation space by the FMPP. A multi-chip construction is easily implemented as same as for an LSI memory. The memory-based structure enables a word of the FMPP to be accessed same as a conventional memory. I/O pins are required for address, data and control. The number of data and control pins is constant at any number of PEs, while the number of address pins is proportional to the log to the base 2 of the number of processors. Thus, total number of IO pins slowly increases as the number of PEs.

SIMD control method. All the PE of the FMPP are controlled by a single instruction. It is an SIMD (Single Instruction Multiple Data stream) machine, where all processors work simultaneously by a single broadcast instruction. The silicon area required by control logics is slightly smaller than MIMD approaches.

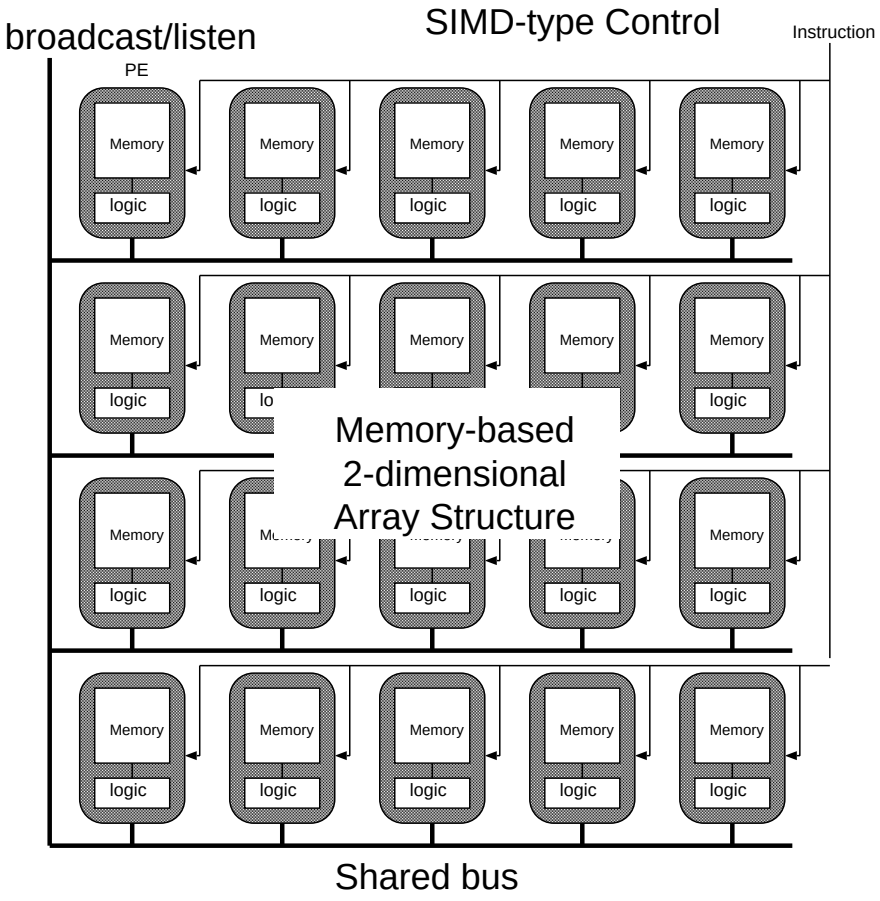


Figure 3.1: Functional memory type parallel processor architecture.

Simple communication network through a shared bus. The shared-bus is the most simple way to connect multiple PEs. It enhances the layout density, while applications on the FMPP should remove inter-processor communication and reduce communication between processors. An outer control logic or CPU can access the content of each word on the FMPP through read/write operations word by word like a conventional memory.

Massively parallel computing on huge number of processors. Memory-based simple structure realizes massively parallel computing. The number of processors can be increased year by year as progress of memory VLSI technology.

Easy to achieve highly dense layout. Processors of today contain too complex circuits and networks. Now, they are semi-automatically implemented by logic and layout synthesizers paying the cost of silicon area. The two-dimensional regular array structure and simple communication network of the FMPP allows highly dense layout. All we have to do is to design a layout pattern of a PE and to put it into array, which can be implemented by interactive manual design strategies.

Low power computing. Chandrakasan mentions that parallel processing must decrease power dissipation[CSB92]. Suppose that two processors work in parallel. The clock frequency of them may be half of that of a single processor if the same through-put rate is assumed. On that condition, the supply voltage can be dropped. The power dissipation of such a two-processor system is 0.36 of that of a single processor system. Thus, the FMPP must decrease the power dissipation considerably. In the Von-Neumann system, data transfer between processor and memory consumes large power. The FMPP also reduces power to perform processing inside memory to decrease communication between memory and a processor.

3.2 FMPP Architectures According to the PE Granuality

Here we discuss three FMPP architectures according to the granuality of a PE: the bit-oriented structure called Bit-Parallel Word-Parallel, BPWP: Figure 3.2(a), the word-oriented one called Bit-Serial Word-Parallel, BSWP: Figure 3.2(b) and the block-oriented one called Bit-Parallel Block-Parallel, BPBP: Figure 3.2(c).

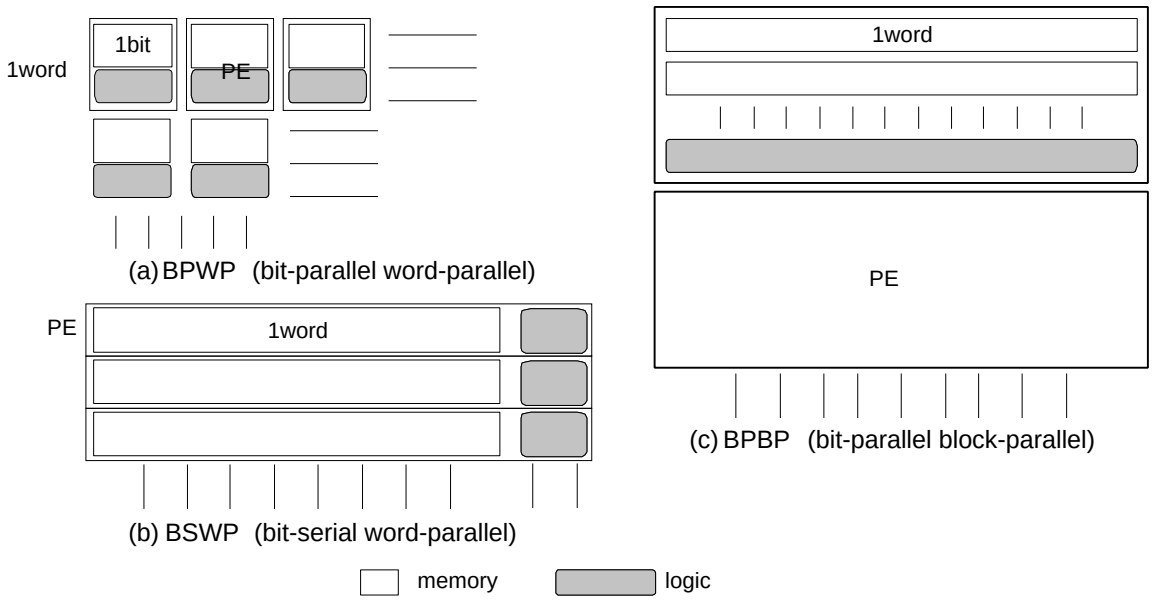


Figure 3.2: Several FMPP architectures according to PE granuality.

In the BPWP architecture, each PE consists of a one-bit memory cell and an ALU. We can expect a large amount of parallel computing in this architecture with the expense of a large amount of hardware required for all the PEs. This is suitable for algorithms which require the same operations on every bit.

In the BSWP architecture, each PE consists of one word of memory cells and a bit-serial ALU. An operation on every word is processed in a word-parallel but bit-serial manner. We can treat a

conventional content addressable memory (CAM) as a BSWP FMPP[YWST91], where each word is considered as a PE. The amount of hardware for a BSWP FMPP is much the same as that for a CAM, thus integration density can be relatively high.

The BPWP and BSWP architectures have the following drawbacks. As for the BPWP architecture, the integration density is not high since the same number of ALUs as that of memory cells are required. The area of each PE should be minimized, which situation makes it difficult to enhance the functionality of the ALU. In the BSWP, the area of a PE is less severe than that of the BPWP. We can realize an ALU with various functionalities. However, computation time is getting longer as the bit width of words increases. Another problem on the BSWP is the lack of ability for inter-word operations such as an addition on two words. If we perform an operation that requires multiple operands in the BSWP, both operands and the result should be stored in a single word and the operation should be performed all the way in a bit-serial manner which consumes much longer processing time than in a bit-parallel manner (See local addition described in Section 3.3.1).

A block-oriented implementation called Bit-Parallel Block-Parallel (BPBP:Figure 3.2(c)) is proposed to achieve both high parallelism and highly dense layout. A PE called a block consists of a group of words and an ALU. A block corresponds to a small processor with several registers and an ALU. It is faster than the BSWP, while the amount of hardware is expected to increase slightly compared with that of the BSWP FMPP. The BPBP architecture merges high parallelism of the BPWP and high density of the BSWP.

The BPBP allows logical and numerical operations on two words. We must carefully define the number of words in a block. If we save both operands and the result at an operation on two words, at least three words should be included in a PE. Too many words in a block may spoil the degree of parallelism. The suitable number of words in a block depends on applications. The more complex operations we require, the more word should be included in a block in order not to spoil the high integration density of the BPBP. As for the BPBP-FMPP introduced in the next chapter, a PE has four words. This is because at least four words are required for numerical operations on two words in the BPBP-FMPP; two words for the operands, one word for the result and one word for the carry. An application specific FMPP called the FMPP-VQ in Chapter 5 has 16 words in a PE which is defined by the dimension of the vector.

3.3 Implementations of the FMPP Architecture

Here, several implementations of the BSWP and BPBP architectures are briefly described. Among these implementations, the BPBP-FMPP and the FMPP-VQ are explained in detail in Chapter 4 and Chapter 5.

3.3.1 Bit-serial Word-parallel Architecture

At the beginning phase of the research on the FMPP architecture, we regard the CAM described in Section 2.2.1 as a bit-serial word-parallel FMPP. The CAM has functionalities of search operation, multiple response resolution and parallel write operation. Of course, we can find words whose contents are matched to a key in the CAM. In addition to that, we can find the minimum or maximum (i.e. extremum) value among all the word. It is done to repeat the search operation from MSB to LSB. The search operation also enables threshold search where all the word above or below some threshold value can be detected. Numerical operations such as addition or subtraction can be done on the CAM owing to its search and parallel-write capability.

Extremum and Threshold Search on the CAM

Let us introduce the procedure to search the minimum value stored in the CAM. Figure 3.3 explains the data flow of the minimum value search using 4bit CAM words. Figure 3.4 shows the procedure for the minimum value search. The value *X* shows the masked bit. The minimum value search repeats the search operations from MSB to LSB. The signal E_R is supplied from the multiple response resolver in the CAM. If it becomes false, the target bit of the key value turns to true(1). The extremum search is done in O(N). The parameter N means the bit width of the CAM word. It does not depend on the number of words.

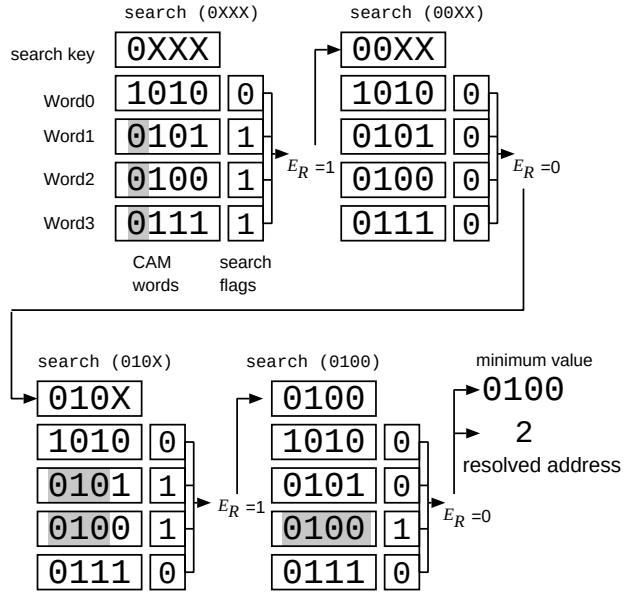


Figure 3.3: Flow chart of the minimum value search.

min=XXXX #(every bit is masked)
for $i = 3$ **downto** 0 # Iteration
  $min[i] = 0$
  search(min)
  if $E_R = 0$ **then**
    $min[i] = 1$
  endif
end

Figure 3.4: Procedure for the minimum value search.

The threshold search is done to iterate the search operation similar to the extremum search. But the logical-OR functionality is required to the search flag, which is not implemented in the conventional

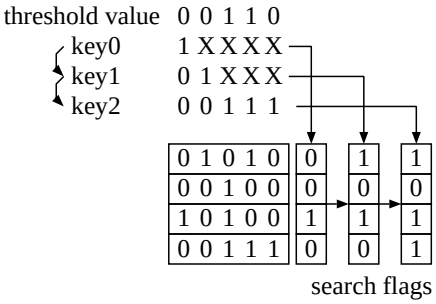


Figure 3.5: Threshold search on CAM.

4kbit CAM[OYN85]. The data flow of the threshold search is schematized in Figure 3.5. The computational complexity is O(N).

Numerical Operations on the CAM.

The CAM can be applied to numerical operations[YWST91]. We introduce two numerical operations. One is parallel addition between an outer value and all the words in the CAM (global addition). The other is parallel addition between all the inner words (local addition). These numerical operations are performed to iterate the search and parallel write operations from LSB to MSB. Figure 3.6 and Figure 3.7 show the procedure and flow chart of global addition. A word (PE) stores an operand A and a carry bit C. An iteration consists of two search operations and two parallel write operations. The procedure and flow chart of local addition are depicted in Figure 3.8 and Figure 3.9. An iteration consists of four search operations and four parallel write operations. A word stores two operands A and B and a carry bit C. These numerical operations are done in O(N).

Functional Memory for Parallel Addition

As in the previous section, the CAM has capability of numerical operations. It has some drawkbacks.

1. Operands should be placed in the same word in local addition.

2. A single-bit computation consists of several search and parallel write operations, which consumes processing time.

To compensate such drawbacks, a bit-serial word-parallel FMPP designed for parallel numerical operations has been proposed, which is called "Functional Memory for Parallel Addition." All the PEs are laid out in a two-dimentional array (Figure 3.10). A PE consists of multiple words and a bit-serial ALU (Figure 3.11(a)). A memory cell is a DRAM cell shown in Figure 3.11(b). The ALU has functionalities of addition and logical operations between two words. It is implemented in a

```
for i = 0 to N − 1
    if B[i] = 0
      if (A[i], C) == (0, 1)        # search
        (A[i], C) = (1, 0)          #parallel write
      else if (A[i], C) = (1, 1)    # search
        (A[i], C) = (0, 1)          #parallel write
      endif
    else if B[i] = 1
      if (A[i], C) == (1, 0)        # search
        (A[i], C) = (0, 1)          #parallel write
      else (A[i], C) = (0, 0)       # search
        (A[i], C) = (1, 0)          #parallel write
      endif
end
```

Figure 3.6: Algorithm for global addition.

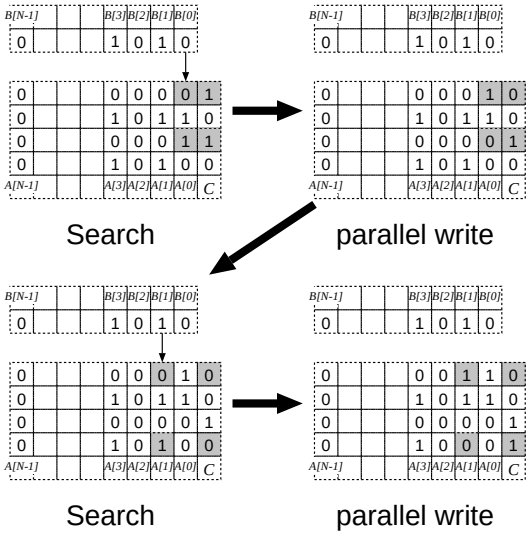


Figure 3.7: Data flow-chart of global addition.

0.5μm CMOS process. Figure 3.12 shows a layout pattern of the PE. The layout pitch of the ALU is exactly matched to the height of the two words. The PE has multiple words, which strucutre may not be in the category of BSWP. But it is cassified to BSWP, since the ALU is bit-serial.

3.3.2 Bit-parallel Block-parallel Architecture

The first FMPP architecture appeared in 1989 [NYT89] has the bit-parallel block-parallel (BPBP) structure for image processing. Here, we call it a functional memory type parallel processor for image processing (FMPP-IP). Figure 3.13 shows its structure. PEs are connected only to the adjacent ones, which does not come under the definition of the FMPP. A PE consists of multiple words and an ALU (Figure 3.14). An ALU complies so-called carry-save adder where addition is done in every bit in parallel, but the carry is propagated bit by bit. Addition is done in a bit-serial manner. In the bit-serial ALU as in the BSWP FMPP, multiplication consumes too many operations in proportion to the square of bit-length (O(N^2)). Multiplication by the carry-save adder, however, can be completed in O(N). The PE of the FMPP-IP is designed with a standard cell library in a 2μm CMOS process. Since the final layout pattern is automatically generated, its area becomes large. They mention that 18 PEs can be implemented in a 1cm^2 die.

This thesis explains two implementations of the BPBP architectures in detail. One is called “Bit-parallel Block-parallel Function Memory Type Parallel Processor (BPBP-FMPP).” Its PE complises four CAM words and a bit-parallel ALU. The other is called “Functional Memory Type Parallel Processor for Vector Quantization (FMPP-VQ).” Its PE consists of 16 SRAM words and a bit-parallel

for $i = 0$ **to** $N - 1$
  if $(A[i], B[i], C) == (0, 0, 1)$
    $(A[i], C) = (1, 0)$
  else if $(A[i], B[i], C) == (0, 1, 1)$
    $(A[i], C) = (0, 1)$
  else if $(A[i], B[i], C) == (1, 1, 0)$
    $(A[i], C) = (0, 1)$
  else if $(A[i], B[i], C) == (1, 0, 0)$
    $(A[i], C) = (1, 0)$
  endif
end

Figure 3.8: Algorithm for local addition.

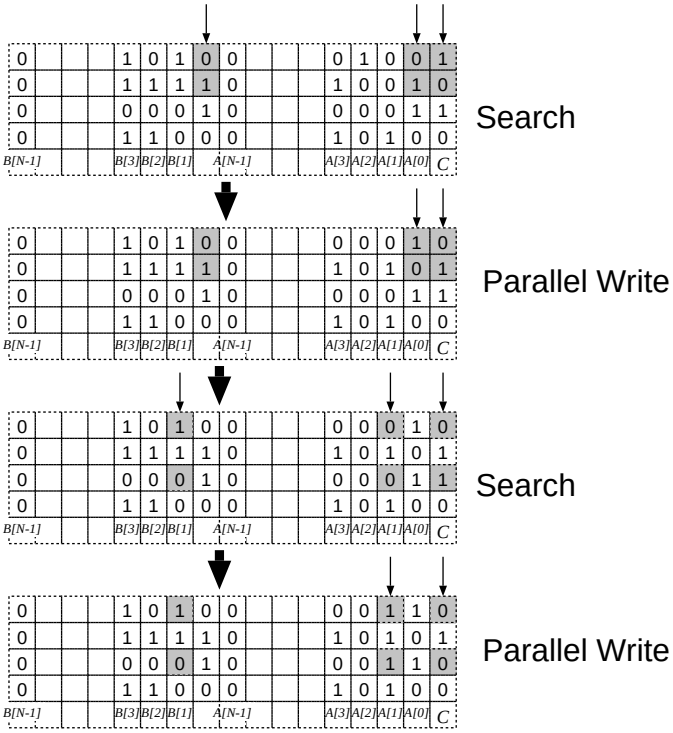


Figure 3.9: Data flow-chart of local addition.

ALU. Both ALUs have a carry-propagate adder which can computes addition in O(1). To achieve highly dense layout, they are implemented in a full-custom method. The layout pitch of the ALU is exactly matched to the width of the word. A PE is implemented in a square region. The PE of the BPBP-FMPP is 32bit wide and has rich functionalities of all the logical operations, addition and multiplication, which enlarges the area of the PE. As the result, An implementation of the BPBP-FMPP has only 8 PEs laid out in a one-dimensional array. In the FMPP-VQ, a PE is 12bit wide and designed for a specific application, “nearest neighbour search” of vector quantization. We can implement 64 PEs laid out in a two-dimensional array.

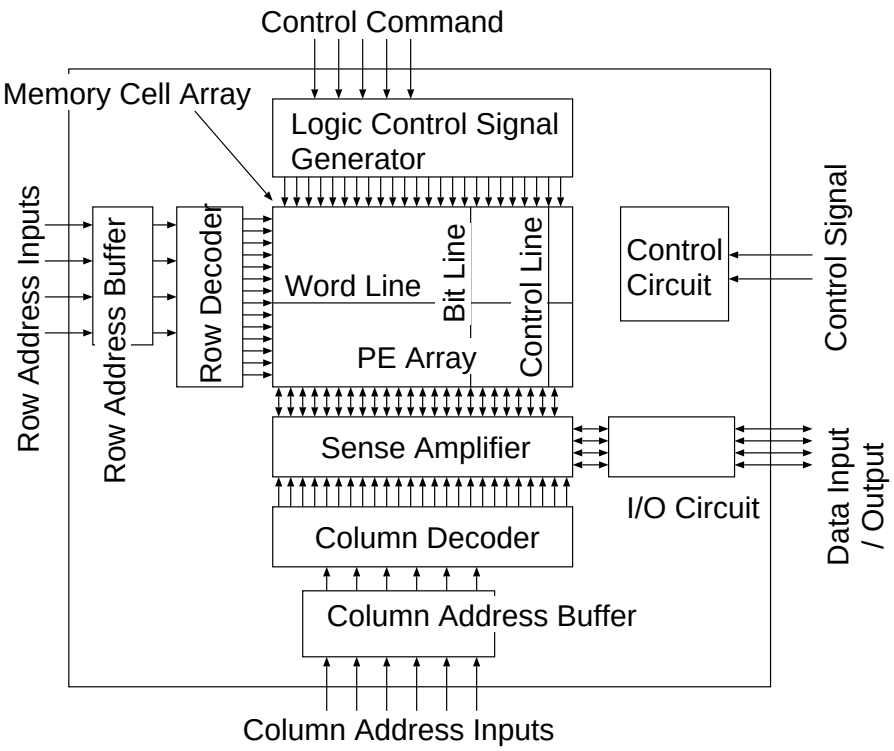


Figure 3.10: Whole structure of the functional memory for parallel addition.

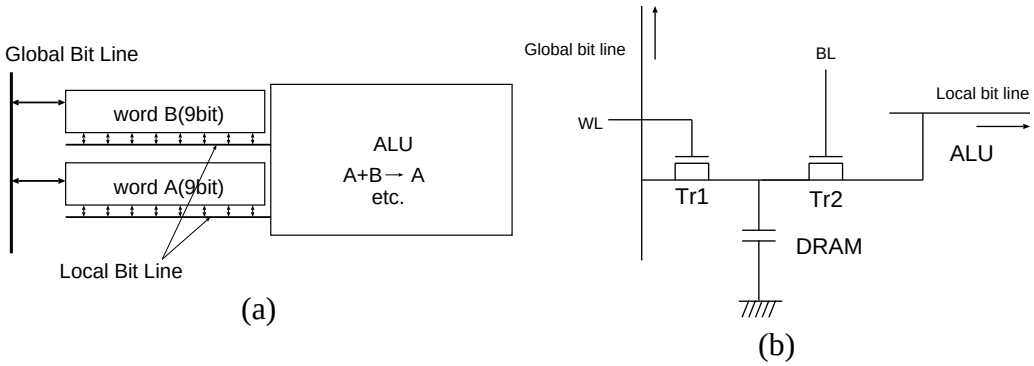


Figure 3.11: Processing element of the functional memory for parallel addition(a), its memory cell (b).

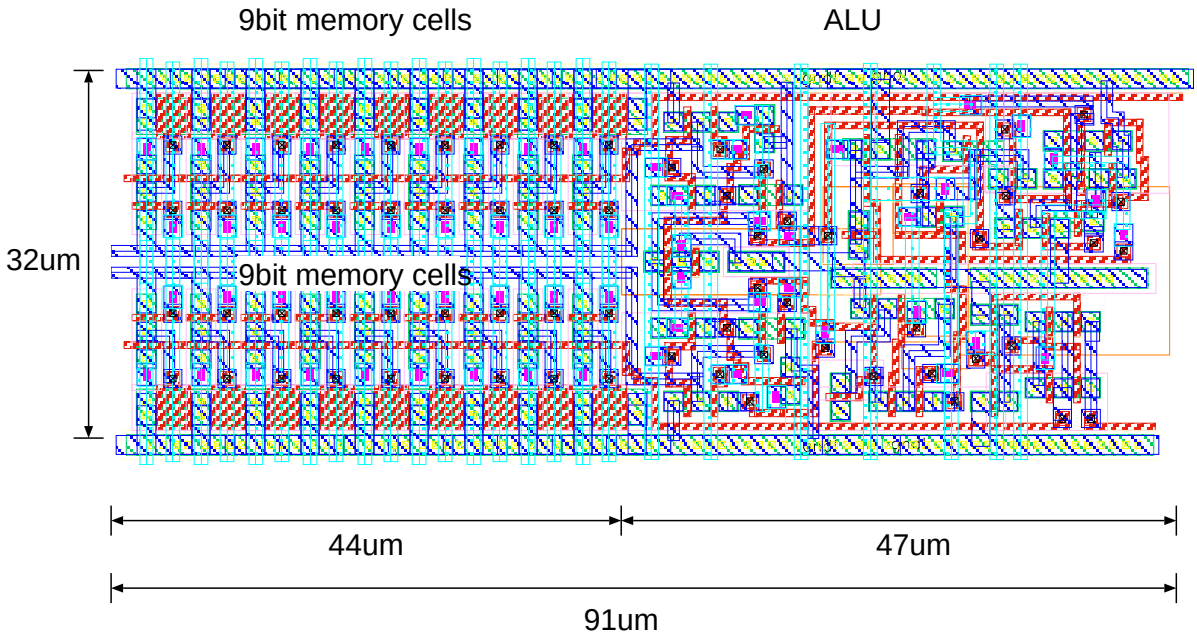


Figure 3.12: Layout pattern of a PE.

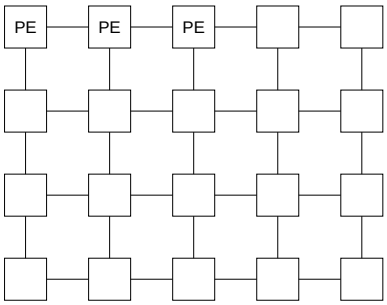


Figure 3.13: Structure of the FMPP-IP.

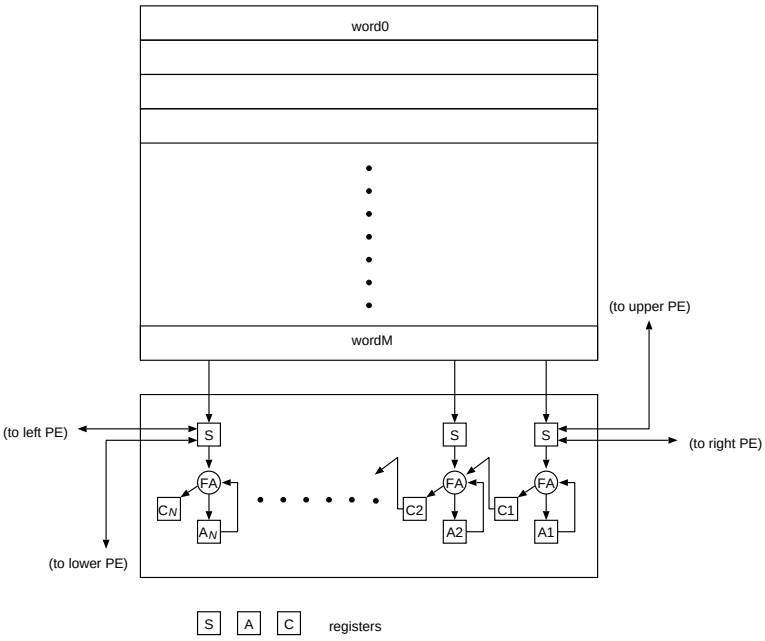


Figure 3.14: Processing Unit of the FMPP-IP.

3.4 Parallel Computation Efficiency on the FMPP

Here we address the parallel computation efficiency on the FMPP compared with a conventional Von Neumann computer in Figure 2.1.

3.4.1 Von Neumann Bottle Neck on the Conventional Computer

As described in Section 2.1.1, the current conventional computer has a Von Neumann bottleneck between CPU and main memory. The Processor-DRAM Gap has been becoming larger and larger as shown in Figure 3.15[Fro98].

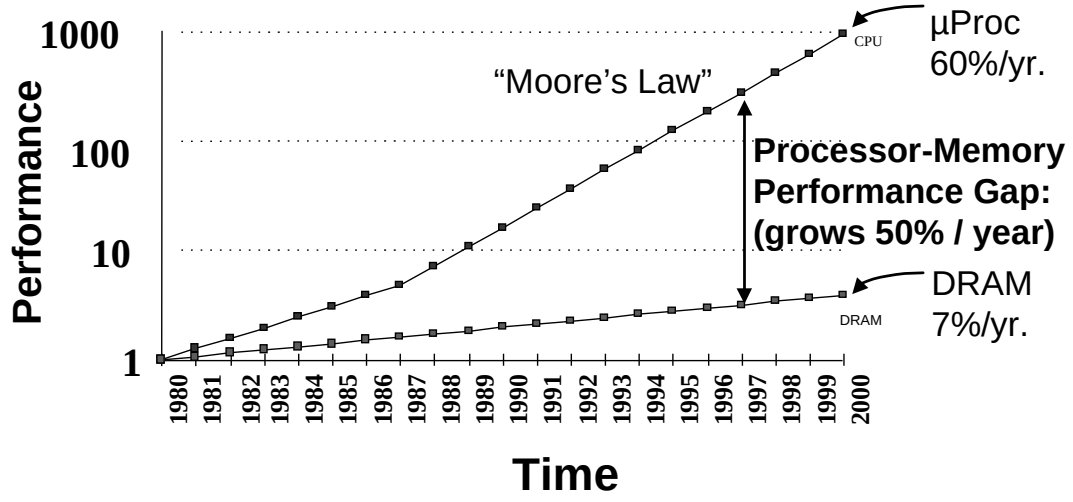


Figure 3.15: Processor DRAM gap[Fro98].

In this situation, the CPU can not display its peak performance as already mentioned in Section 2.1.1. Here, several simulations on a current commercial PC are done to show the Von Neumann bottleneck. Parameters are shown in Table 3.1. The second column values are taken from the PC system Toshiba Portege^{TM} 620CT. The spec of 620CT is described in Table 3.2.

Two programs are executed on the PC. One adds an operand on the main memory with a constant value on a register and stores the result on the main memory (Program A). The other adds two operands on the main memory and stores the result on the main memory (Program B).

Table 3.1: Parameters for a conventional Von Neumann computer.

Name	Value	Synopsis
t_p	10ns.	processor clock cycle
t_a	50ns.	main memory access time
D	64	bus width
B	8∼32	bit width to be processed

Table 3.2: Spec of Portege 620CT.

CPU	Pentium 100MHz	
	Data Bus Width	64
	D-Cache Size	8kB
	I-Cache Size	8kB
Main Memory	EDO DRAM	
	Size	40MB
	Access Time	50ns.
OS	Linux 2.0.32 (a famous UNIX compatible OS for PC/AT)	
C Compiler	gcc 2.7.2.3	
	Option	–O2 (Highly optimized)

$$\text{Program A} \quad : \quad \mathbf{arrayc}[i] = \mathbf{arraya}[i] + \mathbf{const};$$
$$\text{Program B} \quad : \quad \mathbf{arrayc}[i] = \mathbf{arraya}[i] + \mathbf{arrayb}[i];$$

To execute Program B, the CPU should access the main memory three times: twice to load operands and once to store the result. Similarly, Program A accesses the main memory twice. Thus, to use above parameters, total execution time for these two programs are described by Equation (3.1) and (3.2) respectively.

$$\text{[Exec Time of Program A]} = 2 \cdot t_a + t_p = 110\text{ns}. \tag{3.1}$$
$$\text{[Exec Time of Program B]} = 3 \cdot t_a + t_p = 160\text{ns}. \tag{3.2}$$

In Portege 620CT, Program A takes 102ns., while Program B takes 160ns. Note that these are

average values of huge number of iterations. They are almost equal to the values obtained from Equation (3.1) and (3.2). These results clearly show the Von Neumann bottleneck. The processor has capability to complete addition in 10ns, while each data transfer between the processor and the main memory takes 50ns. It is no use to increase the processor clock frequency on the Von Neumann computer. The execution time is limited by the DRAM access time.

3.4.2 Parallel Computation on the FMPP

Figure 3.16 depicts a parallel computing system using the FMPP, which structure is similar to the conventional Von Neumann computer. Some part of the main memory, however, consists of the FMPP. We assume that the FMPP has a capability to perform numerical operations between two words simultaneously in all the PEs. Parameters are defined as follows.

- n number of clock cycles for numerical operations on the FMPP
- t_f FMPP clock cycle

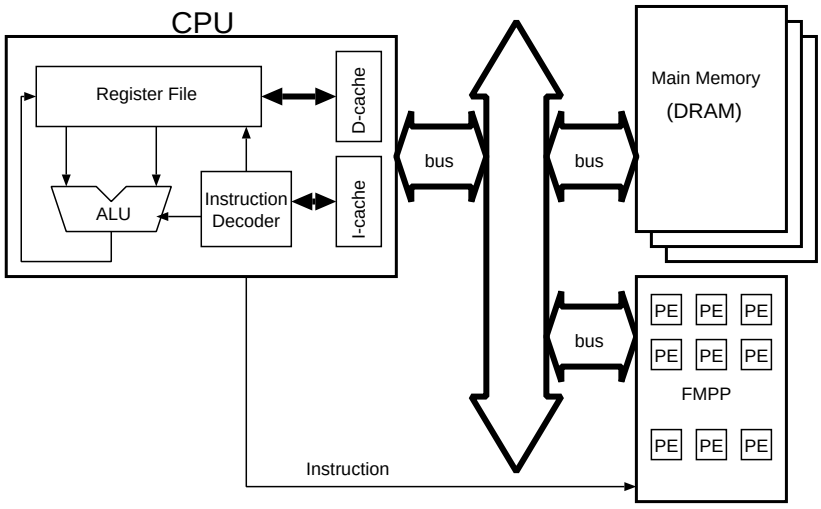


Figure 3.16: A computer system using an FMPP as a part of main memory.

We compare the total execution time on the FMPP system and the Von Neumann system. The operation performs a numerical operation denoted by $*$ to a large number of data M on the memory as follows.

for $i = 0$ **to** $M - 1$
  mem[$i * 3$] = **mem**[$i * 3 + 1$]$*$**mem**[$i * 3 + 2$]
end

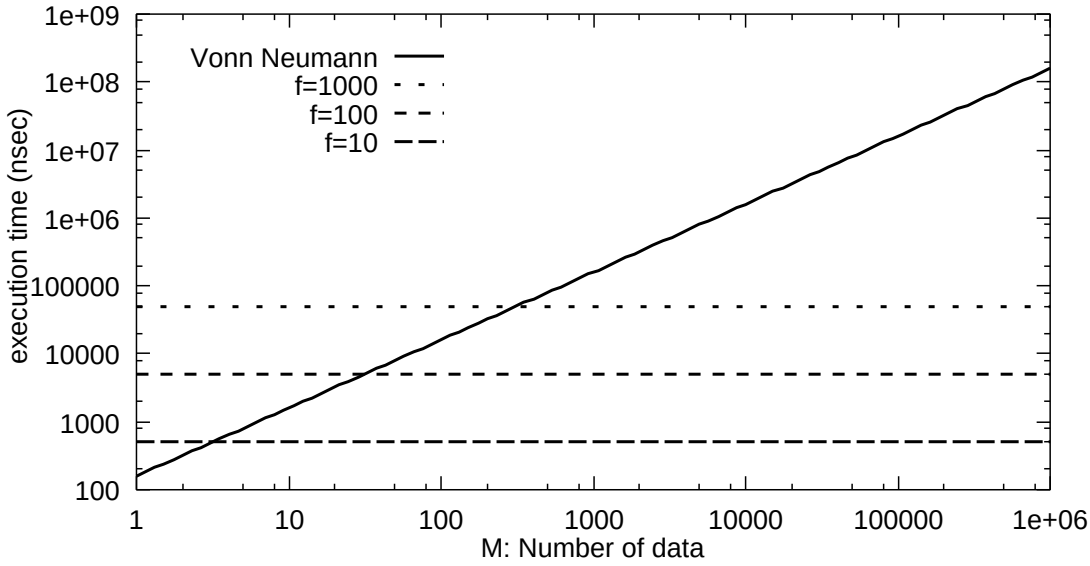


Figure 3.17: Total execution time on the Von-Neumann computer and on the FMPP.

The total execution time on the Von Neumann model for a single iteration becomes $M \cdot (3t_a + t_p)$. The FMPP simultaneously performs the same operation for all M data within the constant time of $f \cdot t_f$. The FMPP gets better performance than the Von Neuman computer even when M is small. Figure 3.17 shows the total execution time for M where t_f is assumed to be 50ns as the same value of the DRAM access time. When $f = 100$ and $M = 1000$, the FMPP outperforms the Von Neumann computer by 32 times. Note that we can neglect the time to prepare data structure on the FMPP. It is becase the same data structure must be prepared on the conventional main memory.

Operations on the FMPP are not always superior to the Von Neumann computers, since all the current commercial CPUs have cache memory. Once data come to cache memory, CPU can quickly access the data. If the CPU executes a group of operations for the data size of which is smaller than the cache size, the CPU can access the data directly from the cache memory. It improves the performance of the Von Neuman system. But once a cache misse occurs, the performance is degraded considerably. Several novel cache architectures have been proposed and developed to decrease the cache miss count.

Here, we compare the performance between a Neumann computer system with a cache memory and an FMPP system. A series of N_p operations is performed to a huge number (M) of groups of data, each of which includes the number d of data. Figure 3.18 depicts two systems. We suppose the following conditions.

1. A CPU can access its cache memory in n clock cycles.

2. The size of cache memory $S_c > d$

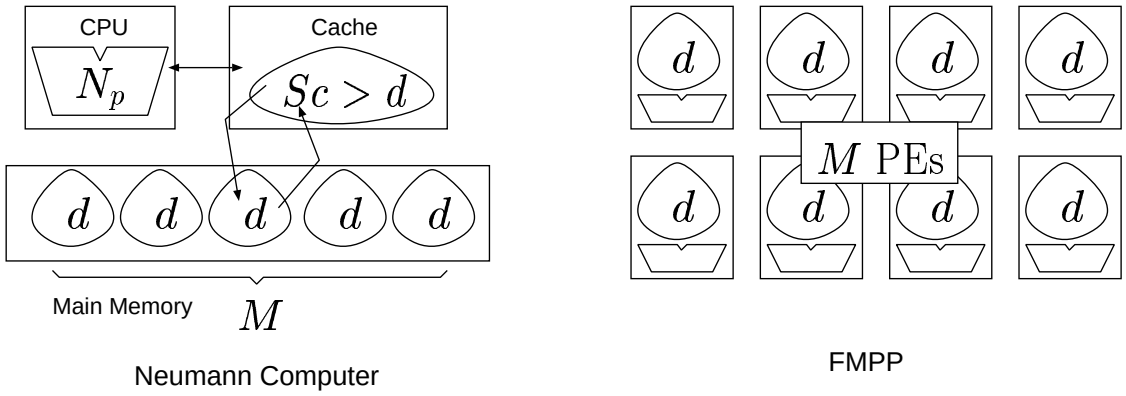


Figure 3.18: Von Neumann system with cache memory and FMPP-based system.

3. The CPU can access all the data from its cache memory besides the first load and the last store operations for the main memory.

4. A PE of the FMPP has d words.

5. The FMPP can complete the same operations in $N_f = f \cdot d \cdot N_p$ cycles.

Under the above condition, the CPU can complete the operations in the following period.

$$\begin{aligned} t_{onCPU} &= t(\text{access and store time for DRAM}) + t(\text{Command execution time on the CPU}) \\ &= 2 \cdot d \cdot t_a + n \cdot t_p \cdot d \cdot N_p \end{aligned} \tag{3.3}$$

On the other hand, the FMPP can complete them f times slower than the CPU as follows.

$$t_{onFMPP} = f \cdot N_p \cdot t_f \cdot d \tag{3.4}$$

The performance efficiency P_{effi} is defined as follows.

$$\begin{aligned} P_{effi} &= \frac{t_{onCPU}}{t_{onFMPPP}} = \frac{(2 \cdot d \cdot t_a + n \cdot d \cdot t_p \cdot N_p) \cdot M}{f \cdot N_p \cdot t_f \cdot d} \\ &= \frac{(2 \cdot t_a + n \cdot t_p \cdot N_p) \cdot M}{f \cdot N_p \cdot t_f} \end{aligned} \tag{3.5}$$

The parameter M denotes the number of PEs of the FMPP. There are so many parameters in Equation (3.5). Some actual values are given in Table 3.3. Using these parameters, Equation (3.5) is simplified as follows.

$$P_{effi} = \frac{t_{onCPU}}{t_{onFMPPP}} = \frac{(5 + 2 \cdot N_p) \cdot M}{5 \cdot f \cdot N_p} = \frac{5 + 2 \cdot N_p}{5 \cdot N_p} \cdot \frac{M}{f} \tag{3.6}$$

Table 3.3: Parameters to compare a Neumann Computer system with an FMPP system.

Parameter	Synopsys	Value
t_a, t_f	DRAM/FMPP access time	$5t_p$
n	Cache access clock count	2

Figure 3.19 shows the performance efficiency according to M/f and N_p. As N_p becomes larger, the performance efficiency asymptotically approaches to $0.4M/f$. It suggests that an FMPP system outperforms the Neumann computer by 40 times with $M = 1000$, $f = 10$. The condition $M = 1000$ means that we prepare 1000 PEs, while $f = 10$ means that the required number of clock cycles on the FMPP can be 10 times bigger than that on the CPU. Note that f is the number of clock cycles. The above condition assumes that the clock cycle of the FMPP is 5 times longer than that of the CPU. If the CPU working at 100MHz can complete the operations within 100 clock cycles (1μs.), the FMPP working at 20MHz must complete the same operations within 1000 clock cycles (50μs.). The FMPP is 50 times slower than the CPU.

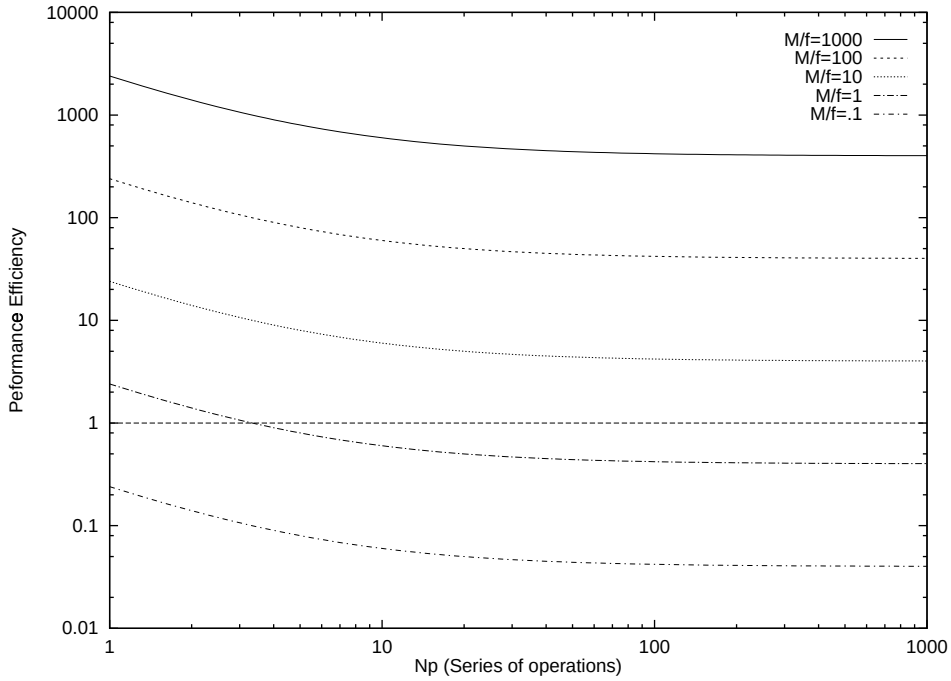


Figure 3.19: Performance efficiency of an Neumann Computer system / an FMPP system. (M=Number of PEs, f=Number of clock cycles on the FMPP system.)

In the above discussion, we assume conditions as follows. On the FMPP computing system, all the operations are done in the FMPP. On the Neumann computing system, all the data can be obtained from the cache memory besides the first load and the last store operations. It takes 2 clock cycles

to access the cache. Actually, some part of required data can be directly obtained from registers. It takes a single clock cycle to access the registers. Thus, the comparison may not be accurate. But, even if all the data could be directly retrieved from registers, the FMPP system with a large number of PEs is superior to the Von Neumann computers.

The current commercial CPU has various functionalities. Almost all operations such as multiplication can be done in a single clock cycle. In the FMPP, however, a high-performance ALU enlarges the PE area and makes the circuitry complicated. In this paper, we introduced several FMPP implementations which have a capability to complete addition of two words within a single clock cycle. Adders can be implemented with a small number of transistors, while multipliers cost too many transistors. The functions of the FMPP must be defined considering a trade-off between performance and circuit area. In the FMPP-based computing system in Figure 3.16, operations are done both in CPU and FMPP. They have to cooperate to complete a job. Operations must be assigned to the CPU or the FMPP so as to minimize the total operation period.

3.5 Summary of the Chapter

In this chapter, we introduce the FMPP architecture. FMPP is a memory-based SIMD shared-bus parallel processor which enjoys the current remarkable progress of semiconductor memory devices. The density of the LSI is doubled every 18 months predicted from the famous Moore's law. The performance gap between the CPU and the memory device, however, becomes larger and larger. The performance of the current computer system is limited by the bandwidth between the CPU and the memory. The FMPP alleviates the performance gap, since operations can be done inside the memory. As mentioned here, if SIMD operations can be done inside a memory, the performance will improve considerably. The performance of the FMPP is linearly improved according to the number of PEs. The structure of the FMPP similar to that of the memory allows highly dense layout. Communication between PEs, however, is limited. Therefore we must choose the suitable operations for the FMPP. The FMPP can be utilized as two ways. One is as a part of main memory on a conventional Von Neumann computer. The other is used as an application-specific processor. In the former style, an FMPP device can work as both main memory and co-processor. In the latter style, an FMPP device work as a processor independently of the CPU.

Here, three FMPP architectures are shown: bit-parallel word-parallel (BPWP), bit-serial word-parallel (BSWP), bit-parallel block-parallel (BPBP). In the BPWP architecture, every PE attached to every bit and word consumes hardware cost. No implementation have been found of the BPWP. The hardware cost of the BSWP architecture is less severe than the BPWP. Lots of associative processors can be found based on the BSWP architecture. A CAM is one of the most popular functional

memory devices. Its word-oriented structure can be regarded as the BSWP FMPP. Extremum search or numerical operations are successfully applied to the CAM. A BSWP FMPP for parallel addition is introduced. It has an ALU for every two words. Here, we mainly focus on the BPBP architecture. It enables operations between two words and operations are done in bit-parallel. We introduce two implementations: the BPBP-FMPP and the FMPP-VQ in the following two chapters. The BPBP-FMPP is designed to utilize as a part of main memory. The FMPP-VQ is developed to implement a low-rate image compression system by vector quantization. It can work independently of the CPU.

The performance efficiency of the FMPP-based system is also discussed here. The FMPP-based system where a part of main memory is replaced with the FMPP shows better performance than the conventional Von Neumann computing system, if the same operations are applied to huge number of data sets. If we can prepare an FMPP with 1000 PEs which has a capability of numerical operations 50 times slower than the CPU, it can perform series of operations 40 times faster than the Von Neumann system. In the FMPP-based system, we must assign operations to the FMPP or to the processor in order to minimize total execution time.

Chapter 4

An Implementation of the Bit-Parallel Block-Parallel FMPP

In this chapter we describe an implementation of the bit-parallel block-parallel FMPP architecture. We have designed and fabricated a prototype LSI[KOT95] BPBP-FMPP based on the BPBP architecture[KTYO93].

The BPBP-FMPP LSI has functionalities of bit-parallel numerical and logical operations on internal two words. Since a CAM cell can execute logical operations on an external data and contents of words, we adopt the structure of a CAM cell as that of a FMPP cell. Using contents of another word as an external data, logical and numerical operations on two words can be performed.

We realize bit-parallel addition in combination with logical operations and a carry propagation using a Manchester carry chain[WE85] which propagates the carry in bit-parallel manner. The structure of a CAM cell enables search operations (content addressing) on the FMPP same as that of CAMs.

Primary operations on the BPBP-FMPP are summarized as follows.

- Bit-parallel block-parallel computations such as logical operations, addition, subtraction and multiplication.
- Search operation.
- Logical operations on flags.
- Parallel write operation.
- Multiple response resolution.

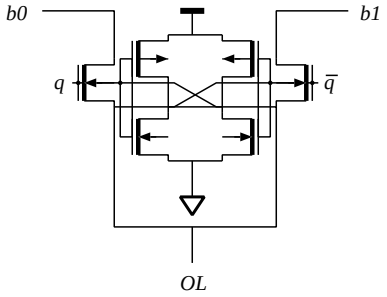
4.1 BPBP-FMPP

The BPBP-FMPP can perform parallel numerical operations on internal two words simultaneously on all PEs. It has various functionalities as a RAM, a CAM and a parallel processor. It performs

addition of two words in a PE in O(1). Multiplication is done in O(m) (m stands for the number of bits of a multiplier). These operations are simultaneously done in every PE. Each PE contains four 32bit words. A single LSI chip contains eight PEs.

4.1.1 Logical Operations on the CAM Cell

We utilize the structure of a CAM cell as that of an FMPP cell, since the CAM cell has a possibility of logical operations on an external operand and contents of words. Logical operations on two words can be done on the CAM cell if a content of another word is given as an external operand as shown in Figure 4.1. Suppose that the CAM cell stores q and an external data is given through *b0* and *b1*. The complemental signals p and $\overline{p}$ from *b0* and *b1* produce $\overline{p \oplus q}$, which is within the original CAM functionality. If one of the bit lines *b0* or *b1* is dropped to the ground level, logical AND ($p \cdot q$) or logical NOR ($\overline{p|q}$) operation can be done.



$b0$	$b1$	OL
p	$\overline{p}$	$\overline{p \oplus q}$
p	0	$p \cdot q$
0	$\overline{p}$	$\overline{p \mid q}$

Figure 4.1: Logical operations on a CAM cell.

4.1.2 Block Diagram

Figure 4.2 shows a block diagram of the 1kbit BPBP-FMPP LSI. There are eight PEs, address IO, data IO and other components such as sense amplifiers or control logics.

Figure 4.3 depicts the structure of a PE, which comprises a memory block, various flags, a multiple response resolver(MRR) and control logics. The memory block is the essential part of the FMPP, where addition, multiplication and logical operations among two words are performed. The number of words in a single PE is four in this implementation, since at least four words should be required for addition: two words for operands, the other two words for temporary values and the result. They are connected through a shared bus inside the PE called the "local data bus." The word in a memory block is hereafter called "an operand word." A memory block comprises four 32bit operand words of FMPP memory cells (w0 $\sim$ w3), two 32bit buffers of SRAMs (P, G) and a carry chain. These SRAMs and the carry chain form an arithmetic logic unit (ALU). The global data bus connect all the memory blocks.

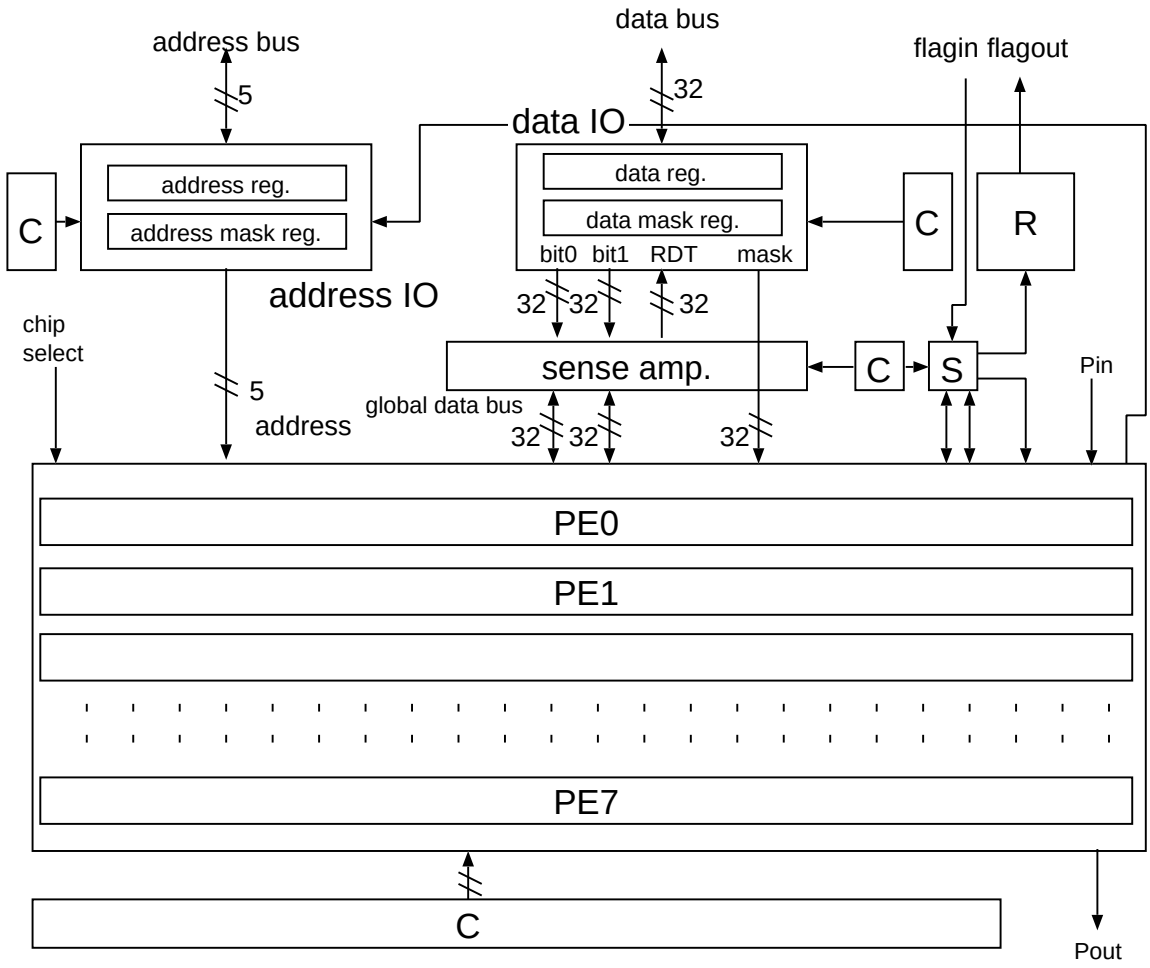


Figure 4.2: Block diagram of the BPBP FMPP LSI.

4.1.3 Primary Operations

Table 4.1 summarizes primary operations on the BPBP-FMPP. In the memory block, addition, subtraction, multiplication and logical operations are available as a parallel processor. They are explained in detail in Section 4.2. As a CAM, the BPBP-FMPP searches for the operand words matched to a given key data, which functionality is called the search operation. The result from the search operation is stored into master or slave flags. The state of these two flags defines a “selected word.” The multiple response resolver(MRR) resolves a single operand word among the selected words, which is called multiple response resolution. The BPBP-FMPP can perform read/write operations similar to conventional RAMs. In addition to that, the parallel write operation is also available. The FMPP writes data to all the selected words, or to multiple words defined with the address masked by the address mask register located at the address IO.

The block flag (BF) prohibits operations of the block on multiplication. The overflow flag(C32)

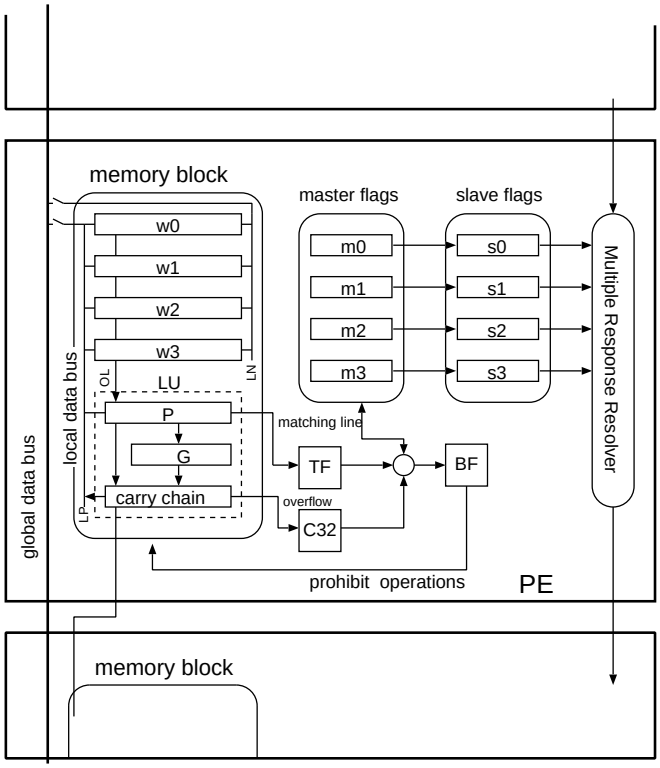


Figure 4.3: Structure of a PE.

stores an overflow value from addition. The temporary flag(TF) temporarily stores a result from the search operation.

The operation called the block transfer is used to communicate between blocks (PEs). A block can communicate with its two adjacent blocks. As for interchip communication, however, the BPBP-FMPP has data and address buses for both input and output in order to decrease the number of IO pins. Processors on separated LSIs should communicate with each other one by one.

4.1.4 Data Mask and Address Mask Operations

Two mask registers, the address mask register (AMR) and the data mask register (DMR) define masks on address and data portions respectively. The mask on the address portion is used to select multiple words. For example, if all bits without LSB are masked, we can choose all the odd or even words by LSB of the address. It is used on the parallel write operation and the search operation.

The mask on the data portion is used in three different ways. One is the mask on the search operations. The masked bits becomes the “don’t care” state. These masked bits are always matched. It is used on the multiplication explained in Section 4.2.5. On the search operation, a key data stored in the DR masked by the DMR is compared with the content of a word in the PE. If the address mask is set, only the specified PEs perform the search operation. The other is used on the partial write

Table 4.1: Primary operations on the BPBP-FMPP.

operation	comment	#step
Operations as a parallel processor in a block		
logical operations	Wi*Wj →Wk	2
addition/subtraction	Wi±Wj →Wk	6
shift/rotate left	Wi << 1 → Wj	5
multiplication	Wi×Wj →Wk	$9m$
Operations as a CAM		
search operation	Results are stored in temporary flags	2
multiple response resolution	Select a single word from multiple selected words.	1
Operations as an RAM		
read/write	Read or write a single word.	1
parallel write	Write multiple words in parallel. selected words or address mask registers define target words.	1
partial write	write data only to the specified bits.	2
Communication between blocks		
block transfer	transfer data in a word into the upper or lower block.	2

$*$ is one of logical AND, NAND, OR, NOR, XOR, XNOR operations.

$i, j, k \in 0 \sim 3$.

m denotes the number of bits.

operation. The content of the masked bits does not change on the partial write operation.

4.1.5 Detailed Structure of the Memory Block

The detailed schematic diagram of the memory block is shown in Figure 4.4. It shows a four-bit slice of the memory block including four operand words (W0∼W3), two buffers (P and G) and a carry chain connected through the local data bus (LP and LN). The detailed structure of each component is discussed here.

Figure 4.5 shows a memory cell of the operand word, which is designed based on a conventional CAM cell[OYN85]. When the data shown in the right side of Figure 4.5 are given, we can get results on OL through three logical operations such as XNOR (exclusive-nor), AND and NOR. In the memory block, one word provides a key data through Tr2 and/or Tr3 and another word receives the data through Tr0 and Tr1. Figure 4.6 shows two buffers P and G. They receive the result from OL,

and then rewrites it to any word in a memory block through LP. A four bit slice of the Manchester carry chain is shown in Figure 4.7[WE85]. It propagates a carry in bit parallel from P and G for numerical operations.

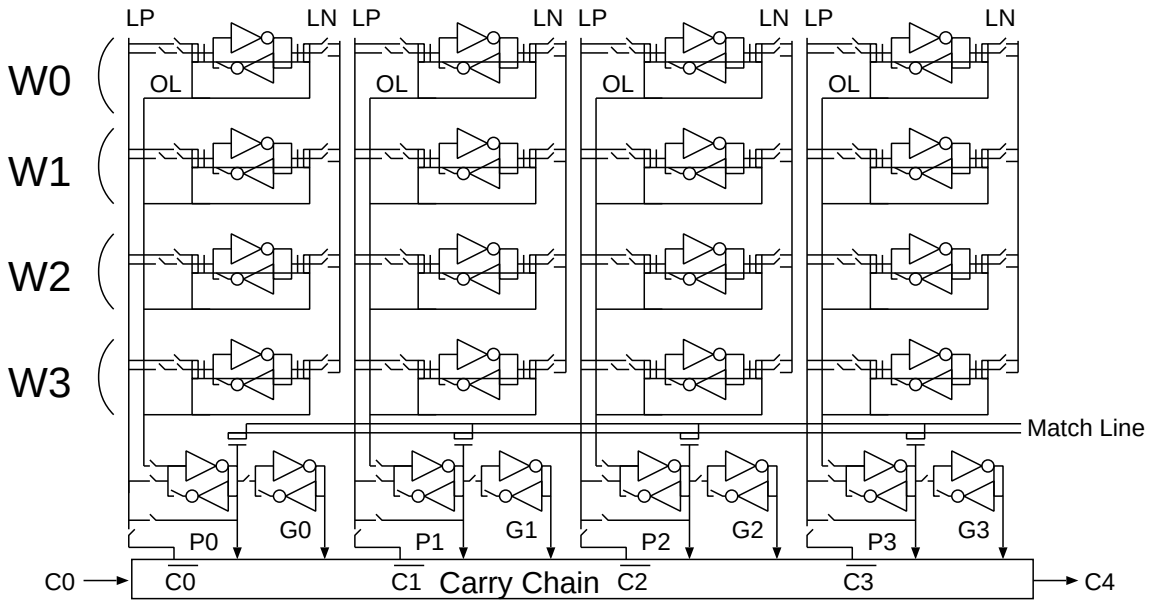
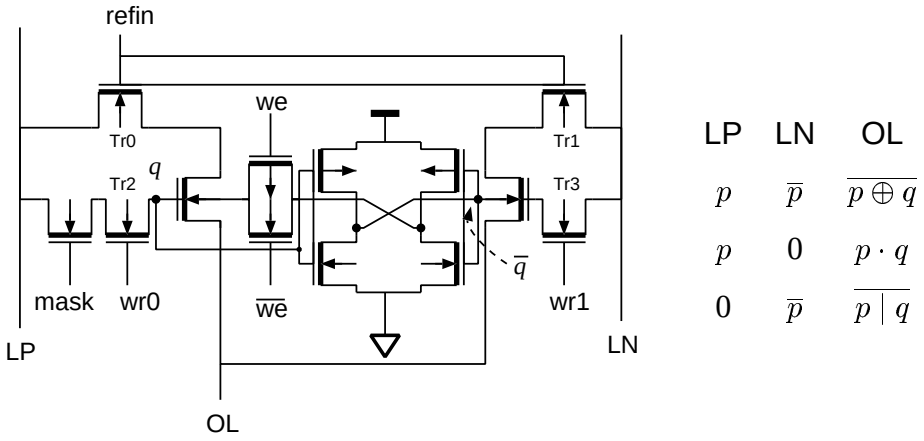


Figure 4.4: Detailed schematic structure of a memory block.



LP	LN	OL
p	$\overline{p}$	$\overline{p \oplus q}$
p	0	$p \cdot q$
0	$\overline{p}$	$\overline{p \mid q}$

Figure 4.5: An FMPP memory cell and logical operations.

4.2 Detailed Operation Strategies

Here, operaion strategies on the BPBP-FMPP are described in detail, such as logical operations, addtion, search operation, multiplication and multiple response resolution.

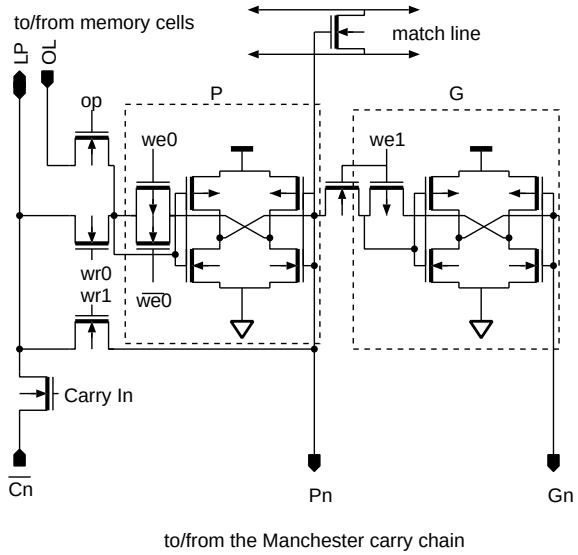


Figure 4.6: The buffers P & G.

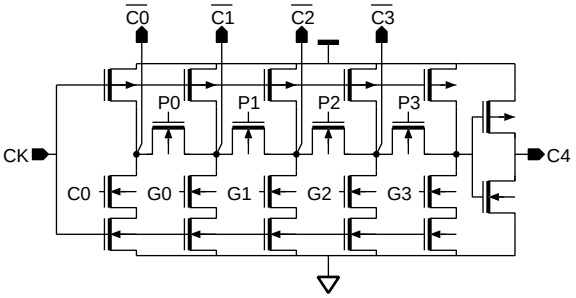


Figure 4.7: Manchester carry chain.

4.2.1 Logical Operations

The BPBP-FMPP has functionalities of all available logical operations such as AND, NAND, OR, NOR, XOR and XNOR between any conbination of two operand words. The result can be written in any operand word.

The XOR operation w0⊕w1→w2 is done in two steps as follows.

Step 1. w0 ⊕ w1 → P

Step 2. P → w2: through wr1 in P.

On the XNOR operation, the signal wr0 in P is activated instead of wr1 at Step 2. Then the complemental value is written to w2. Other logical operations can be done to fall one of local data bus into the ground level accoding to Figure 4.5.

4.2.2 Addition and Subtraction

In the BPBP-FMPP, numerical operations are performed by the comibination of logical operations and a carry propagation accoding to Equations (4.1)-(4.4).

$$C_i = G_i + P_i \cdot C_{i-1} \tag{4.1}$$

$$G_i = A_i \cdot B_i \tag{4.2}$$

$$P_i = A_i | B_i \tag{4.3}$$

$$S_i = C_{i-1} \oplus P_i \tag{4.4}$$

The Manchester carry chain as shown in Figure 4.7 propagates the carry($\overline{C}$) from the carry propagate ($P = A \oplus B$) and the carry generate ($G = A \cdot B$) in parallel. Note that A and B denote two operands of numerical operations. The result Sum is given as $\overline{P} \oplus \overline{C} (= P \oplus C)$. Four FMPP memory cells are required so as not to destroy two operands stored in two operand words. The other two words temporarily store P and $\overline{C}$, and one of them finally stores Sum. The two buffers P and G store P and G respectively. Eight 4-bit Manchester carry chains are connected in serial to propagate a 32bit carry. We use no carry-lookahead scheme, since complex wires from the carry-lookahead unit may break the regularity. Thus, the operation for propagating carry must be critical. Carry-propagation time is derived as 170ns. from the worst-case simulation. To shorten the time for addition, carry propagation and another operation are done simultaneously. Addition is done in six steps as follows (see Figure 4.8).

Initial condition: w0 stores A. w1 stores B.

Step 1: Produce $G = A \cdot B$ and store it to G. G= $A \cdot B$

Step 2: Produce $P = A \oplus B$ and store it to P. P= $A \oplus B$

Step 3: Write P into w2. At the same time, carry propagation is done in the carry chain. w2= P

Step 4: Store $\overline{C}$ into w3. w3= $\overline{C}$

Step 5: Perform XNOR operation between w2 and w3. The result is written to P. P= $(A \oplus B) \oplus \overline{C}$

Step 6: The result in P is written to w2. w2= $\overline{(A \oplus B) \oplus \overline{C}} = P \oplus C = Sum$

Addition takes 1200ns, since the developed LSI works at 5MHz clock frequency.

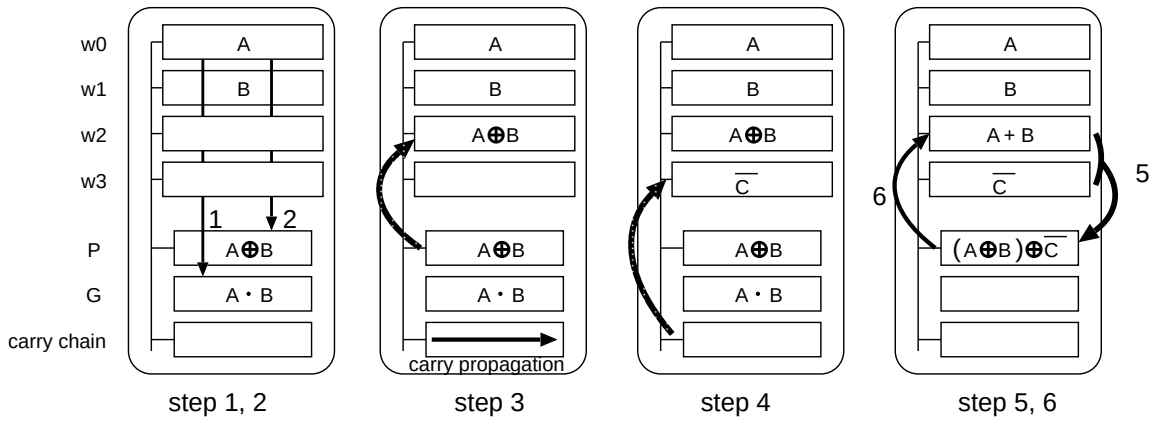


Figure 4.8: Addition between 2 words.

4.2.3 Shift/rotate Left Operation

The shift/rotate left operation shifts one of the operand word one-bit left. The carry chain propagates the target value to one-bit left. The shift left operation are done according to the following procedure.

Initial Condition: w0 stores A

Step 1: G=0

Step 2: w0$\rightarrow$P

Step 3: carry propagation

Step 4: $\overline{C}$ $\rightarrow$w0 (w0 becomes $\overline{A << 1}$.)

Step 5: w0$\rightarrow$P

Step 6: $\overline{\text{P}}$ $\rightarrow$w0

To perform the rotate left operation, two carry propagations are done. At the first carry propagation, the overflow value of the carry is stored in C32. It is used as the LSB of the carry on the second carry propagation.

4.2.4 Search Operation

On the search operation, the XOR operation between a key value broadcast to the PE and an operand word (w0$\sim$3) is done. The result is stored to the buffer P. If the key value is matched to the operand word (described as the matched state), all bits in the buffer P become true, which is equivalent to that the node Pn in Figure 4.6 becomes the ground level. If the key value is not matched (described as the unmatched state), the unmatched bits become false. The two match lines are precharged prior to the search operation. When one of the match lines is fell to the ground level, the other match line keeps the precharged level on the matched state, while it is discharged on the unmatched state. The temporary flag receives the result from the match line, which is connected to the master flags and the block flag. To obtain matches of all the operand words, we should repeat the search operations four times. The results are stored in the master flags.

4.2.5 Multiplication

Multiplication can be done by accumulation of the search operations, additions and shift-left operations. The following procedure shows w0$\times$w1 $\rightarrow$ w2. Figure 4.9 also shows flow of the multiplication.

Initial condition: w0 stores A, w1 stores B. w2$= 0$.

Stage 1: Set $i = m$ and w2=0. (m denotes the number of bits of the multiplier A.)

Stage 2: Perform the search operation to search whether the ith bit of the multiplier B in w1 is 1 or not. Set the BF to 1 if the ith bit is 1.

Stage 3: Add the multiplicand A in w0 to a partial product P_p in w2. The result is stored to w2. Addition is prohibited in the PE whose BF is 0.

Stage 4: Set i to $i - 1$. If i = 0, then halt the procedure.

Stage 5: The shift left operation shifts P_p in w2 to one bit left in every PE. Return to **Stage 2**.

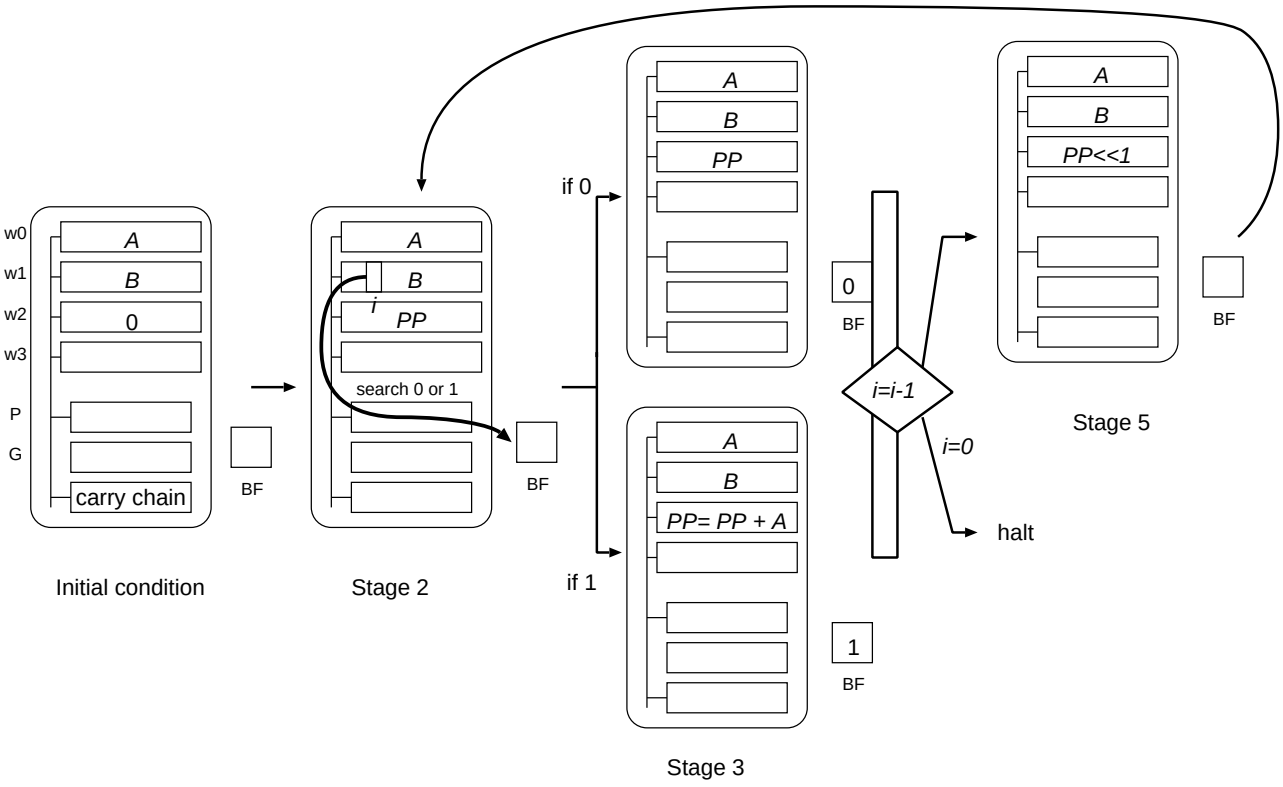


Figure 4.9: Multiplication between 2 words.

Multiplication takes $9 \times m$ steps for the m-bit multiplier. For example, 16bit multiplication takes 144 steps, i.e. 28.8 μsec.

The 1kbit BPBP-FMPP performs 6M(10^6) additions and 280k(10^3) multiplications per second. The performance is not enough, since the developed LSI has only eight PEs. This is mainly because we use an old-fashioned 1.2μm CMOS process and the die size is small. The current sub-micron process gives sufficient number of PEs in a single LSI.

4.2.6 Multiple Response Resolution

In the BPBP-FMPP, multiple response resolution (MRR) is available like CAM. In CAM, the search flag and the garbage flag defines the word to be resolved. A word is resolved if its search flag is true and its garbage flag is false. In the BPBP-FMPP, any combination of the master and slave flags can define a word to be resolved. A resolved word is called a selected word as mentioned before. The BPBP-FMPP can resolve the top-most word among all the selected word. The BPBP-FMPP has the functionality of logical operations to perform the extremum search described later in Section 4.4.1.

4.3 1kbit BPBP-FMPP LSI

Here, we give an overview of the 1kbit BPBP-FMPP LSI, evaluate its layout density and show some test results.

4.3.1 LSI Overview

The BPBP-FMPP LSI has already been fabricated using a 1.2μm CMOS process. Figure 4.10 shows the layout pattern of a four-bit slice of the memory block. It is implemented in a rectangle region, since the layout pitches of all the components are exactly same. Table 4.2 shows an overview of the LSI, which contains 1kbit(32bit$\times$32word) memory cells on a 43 mm^2 die and achieves 5MHz clock rate. In order to enhance flexible control schemes for the first fabrication, most of the IO pins provide primitive control signals. Its die micro photograph can be seen in Figure 4.11. Eight PEs occupy over 80% of the core area except for IO PADs. The memory block is located at the left side of the PE. Memory blocks, master flags and sense amplifiers are implemented with the full-custom design method in order to achieve high density and optimizing performance. The other components such as slave flags and IO registers are implemented with standard cells to enhance productivity although paying a penalty in area. The power dissipation is 100mW when no operation is done, that is, only clock signals are provided. The maximum power dissipation is 300mW when the parallel write operation writes all 32 words in parallel.

The 1.2μm 1poly 2metal CMOS process is equivalent to the process used for a 256kbit SRAM[SSN$^+$90]. Table 4.3 summarizes component areas in the memory block together with the cell area of a 256kbit SRAM. Compared with a 256kbit SRAM, the memory cell area of the BPBP-FMPP is 60 times bigger than that of the SRAM cell. Using a 0.5μm CMOS technology for a 4Mbit SRAM, the area of a memory block becomes 4756μm^2, which value is derived by shrinking the process from 1.2 to 0.5 μm. Including the area for flags and the multiple response resolver, 200 blocks (25kbit) can be integrated on 60mm^2 area. The die size including peripheral circuits and IO PADs will be 75mm^2. Such an FMPP should be faster than the 1kbit FMPPs because the transistor size becomes

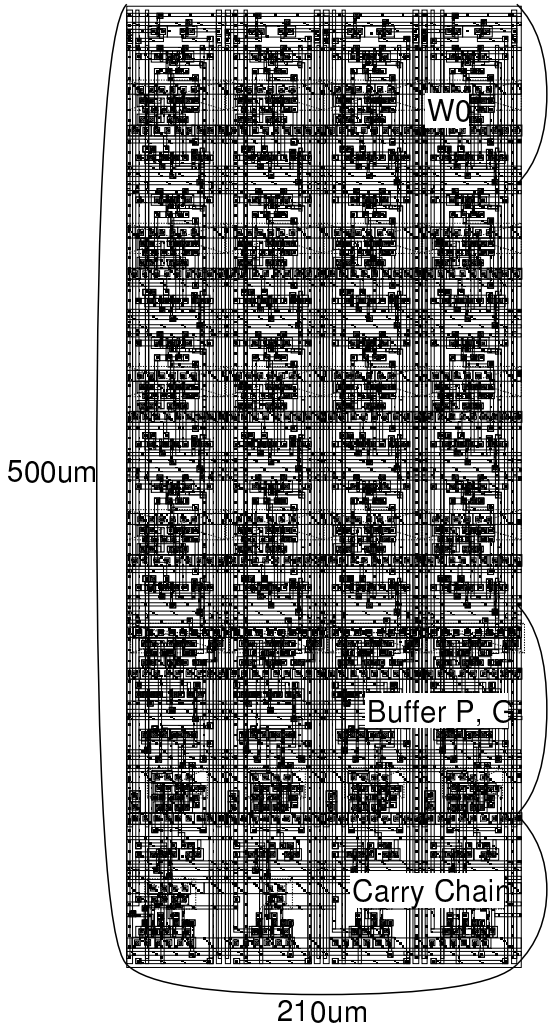


Figure 4.10: Layout pattern of a four-bit slice of the memory block.

smaller. We can estimate that at least the 25kbit FMPP works at 20MHz clock rate. The 1kbit BPBP-FMPP performs 6M(6×10^6) additions per second, while the 25kbit FMPP working at 20MHz clock rate performs 600M additions per second. These performances are derived by a single LSI. If a 1Mbit FMPP can be prepared, it performs 24G additions per second. Such a highly integrated FMPP can be achieved in the future, since the integration density of the FMPP will increase proportionally according to innovations in the LSI process technology.

4.3.2 Test Results

We have been running lots of test sequences using an LSI tester HP82000^{TM}. Almost all operations can be performed correctly. We, however, found a critical fault that one of the local data bus (LN in Figure 4.5) does not work correctly. The memory cell can not pull down LN to the ground. Thus, no logical operation except *AND* operation can be performed. At present, the source of the fault can not be detected.

Table 4.2: Overview of the 1kbit BPBP-FMPP LSI.

Area	43.5mm^2	(5824μm$\times$7420μm)
Operating frequency	5MHz	
IO Pins	134	(8 pins for power)
Package	QFP160	
# Transistor	59000	
Power dissipation	100mW	(nop)
	300mW	(parallel write)

Table 4.3: Component areas of the 1kbit FMPP LSI together with a 256kbit SRAM.

	area(μm^2)	w$\times$h(μm)	# Transistor
1bit slice of a memory block	27645	53.4$\times$517.7	79
averages per one word	6889	53.4$\times$129.0	19.8
a memory cell	4133	53.4$\times$77.4	13
P & G, etc.	7561	53.4$\times$141.6	20
1bit slice of a carry chain	3551	53.4$\times$66.5	7
1bit memory cell of a 256kbit SRAM	109	8.5$\times$12.8	6

Figure 4.12 shows operating waveforms of read/write operations. It repeats read and write operations. The minimum access time is derived as 80nsec. Another test pattern shows that the critical signal path, 32bit carry propagation takes 175nsec., which is almost the same with the value obtained by the circuit simulation.

4.3.3 Comparison for the Circuit Areas between CMOS and CPL Logics

The structure based on the CAM and CPL (Complementary Pass-transistor Logic) of the BPBP-FMPP reduces an area for logical operations between two words compared with that based on the CMOS logic. The former requires only four FMPP cells, while the latter requires four SRAM cells and CMOS logic gates for XOR, AND and OR. Table 4.4 shows the area, the number of transistors and signal delay for each structure. Note that we exclude 32 transistors from the number of transistors, which both structure requires for four 8-transistor SRAM cells. The area for the CMOS structure is twice as large as that for the CAM structure. As for the transistor count for logical operations, the CMOS structure is three times bigger than the CAM structure. Moreover, the CAM structure

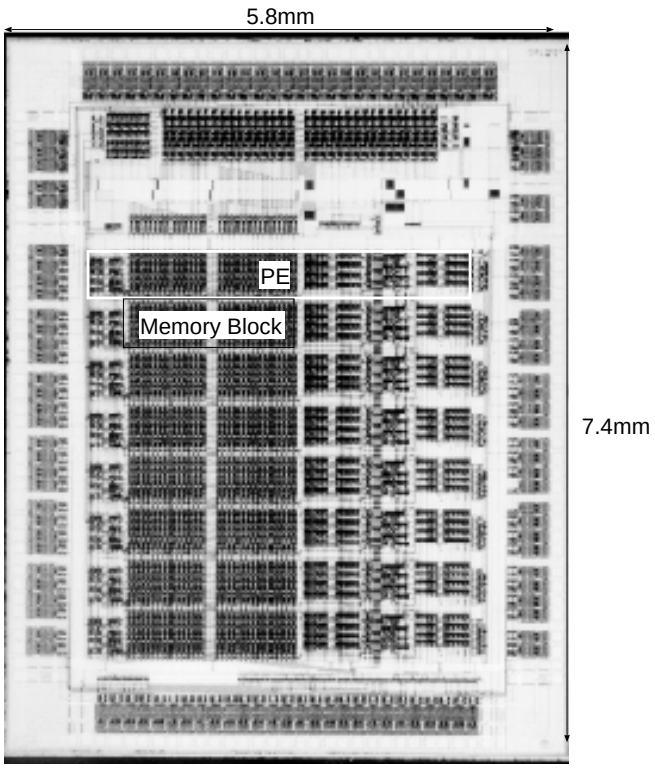


Figure 4.11: Chip micro photograph of the BPBP-FMPP.

performs logical operations faster than the CMOS structure.

structure	area(μm^2)	# transistor	signal delay(nsec.)
CPL(CAM)	34,800	20	15.8
CMOS	68,900	60	19.8

Table 4.4: Comparison of the structure for logical operation.

4.4 Applications of the BPBP-FMPP

Since all the PEs of the BPBP-FMPP work together, it is hard to use the bit line as a communication path between processors. It is only used for the communication between processors and a host. Therefore algorithms on the FMPP should remove inter-processor communication. As for the BPBP-FMPP, it is suitable for the algorithms which require the same operations on every word or every two words among a large amount of words. We evaluate the performance of the BPBP-FMPP for a few applications in comparison with the performances obtained by sequential (word-serial) implementations on an engineering workstation (EWS: cycle time 25nsec.). We evaluate the cycle

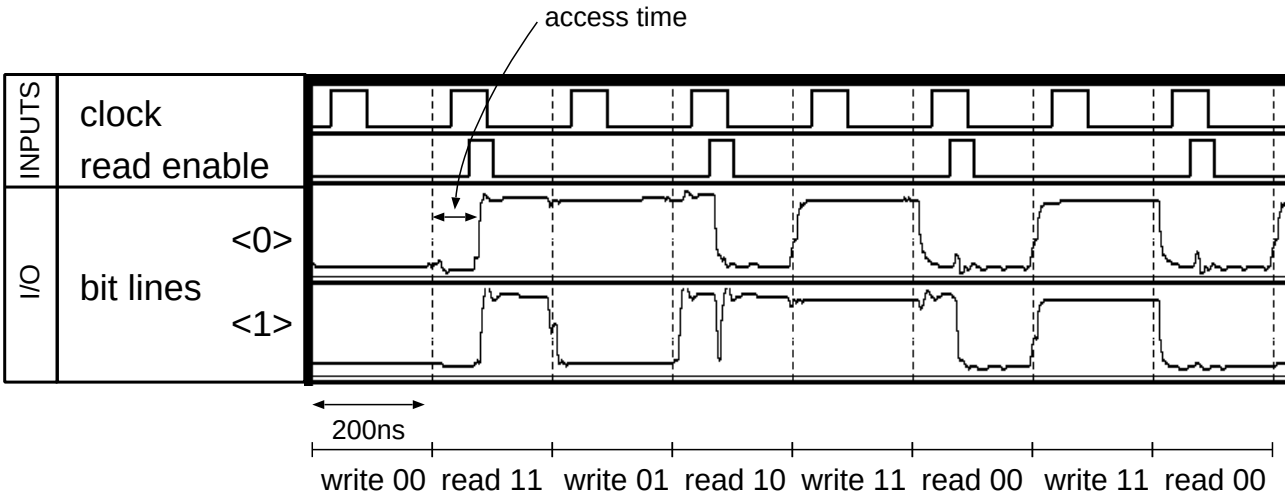


Figure 4.12: Operating waveforms from read/write operations.

time of the BPBP-FMPP is 200ns. from circuit simulations.

4.4.1 Threshold Search and Extremum Search

The threshold search and the extremum search on the BPBP-FMPP are shown, which are frequently used in many applications.

In the threshold search contents of a word w are compared with an external data d. In the CAM-based BSWP FMPP, the threshold search is implemented to repeat search operation from MSB to LSB (See Section 3.3.1). The FMPP can performs the same operation to subtract d from w. If w is less than d, the C32 becomes true. Then, its master flag receives the value from the C32. The MRR detects all the words which are less than d. The FMPP can perform the threshold search in a bit-parallel manner, while an implementation of CAM should be in a bit-serial manner.

The extremum search finds the maximum or minimum value among all words. It is done to repeat search operations and the multiple response resolution from MSB to LSB, which is done in bit-serial similarly to the CAM implementation. Figure 4.13 shows the computation time of the BPBP-FMPP in comparison with that of a sequential implementation. Both two computation times are quite similar. In the developed BPBP-FMPP, response time of the multiple response resolver increases in proportion to the word count, which linearly increases the computation time. An ideal FMPP can solve the problem in constant time, where the multiple response resolver work in constant time at any number of words. It is important to implement a fast multiple response resolver to accelerate the extremum search. We implement a fast response resolver on the FMPP-VQ described in Chapter 5.

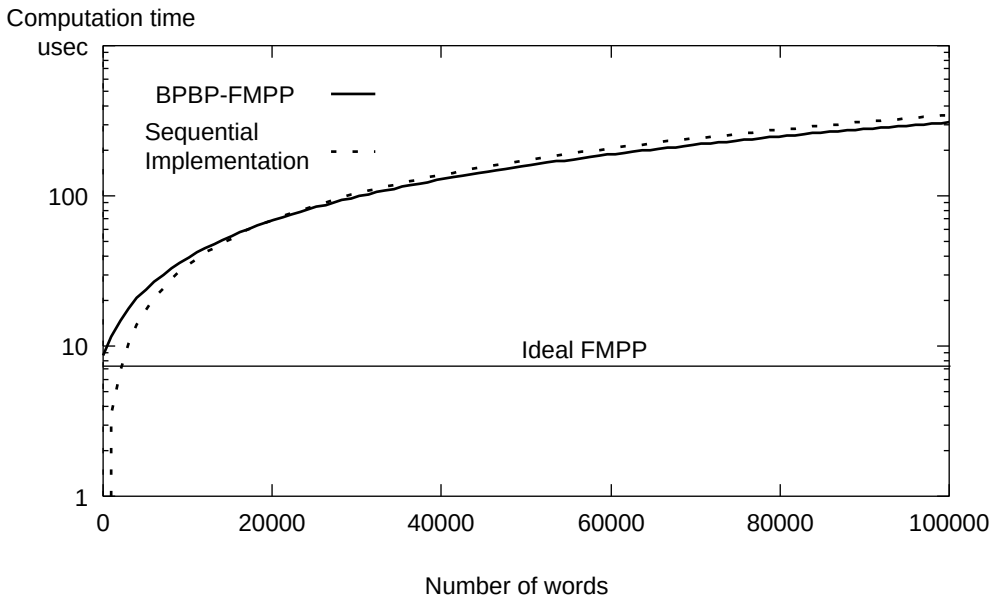


Figure 4.13: Computation time of extremum search.

4.4.2 Knapsack Problem

Knapsack problem is one of typical NP-hard combinatorial optimization problems . We should choose the combination of luggage with maximum sum of profits under the constraint that sum of weight is within a limit. We assume n as the number of luggage, p_n as the profit of the nth piece of luggage, w_n as the weight of the nth piece of luggage and c as the limit of the weight. The algorithm for a BPBP FMPP is developed from that of a BSWP FMPP which is called parallel exhaustive search[YTT88]. The idea of the parallel exhaustive search on the BPBP FMPP is as follows. We assign every possible group to each PE (block) one by one. The weight w_i and the profit p_i of the nth piece of luggage are broadcast using parallel write operation and block j which should contain the nth piece of luggage compute $W_j(= \sum w_i)$ and $P_j(= \sum p_i)$ using parallel addition. The comparison of W_j with c is the threshold search. We can obtain the maximum P_j using the extremum search.

Computation time does not depend on the number of luggage except for the operations to prepare data structure on the BPBP FMPP. It completes within n parallel write operations. Figure 4.14 shows the computation time of knapsack problem on the BPBP FMPP, on a BSWP FMPP and on an implementation of sequential exhaustive search. The BPBP FMPP is 130 times faster than the BSWP FMPP, and 100,000 times faster than the sequential implementation at 20 pieces of luggage. The FMPP can solve knapsack problem much faster than the sequential approach. But huge number of PEs should be prepared, which drawback is common to both BSWP and BPBP FMPPs. If we want to solve a problem with n pieces of luggage, we must have 2^{n+2} words.

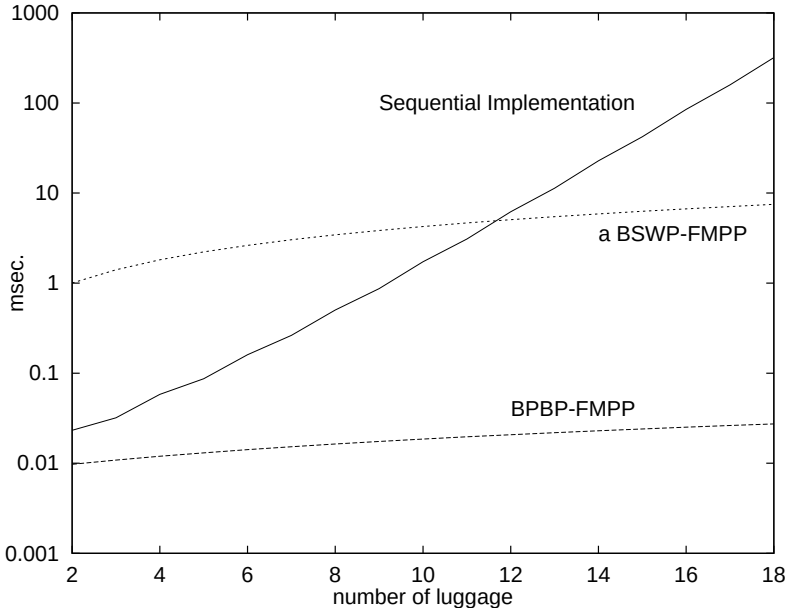


Figure 4.14: Computation time of knapsack problem.

4.5 Summary of the Chapter

We propose a bit-parallel block-parallel(BPBP) FMPP in this chapter. The BPBP-FMPP LSI has been designed and fabricated in a 1.2μm CMOS process. It contains eight PEs implemented in the area of 43.5mm^2. A single PE contains four 32-bit operand words. It can perform numerical operations such as addition and subtraction between any combination of two operand words in bit-parallel. Such numerical operations are done by the combinations of logical operations and a carry propagation. The pass-transistor logics in the ALU decreases the circuit area considerably. We estimate that the required area of our implementation is half of that of a conventional CMOS implementation. The FMPP is a memory-based SIMD shared-bus parallel processor. Therefore, we can easily get highly dense layout because of its two-dimensional regular array structure.

We have been running lots of test sequences using an LSI tester. The 32-bit carry propagation which is supposed to be a critical operation takes 175nsec. Thus, the BPBP-FMPP operates correctly 5.7MHz. Unfortunately, a critical fault was found in a memory cell. Detailed tests can not be continued.

A single PE consists of a bit-parallel ALU and a group of words, which structure enables operations between words. Numerical operations can be completed in O(1), while in the bit-serial word-parallel implementation it takes O(bit-width). The threshold search can be done in O(1) using the numerical operation capability. A famous NP-hard problem, knapsack problem is suitable for the BPBP-FMPP. The BPBP-FMPP can solve knapsack problem of 20 luggage 100,000 times faster than the sequential approach.

The BPBP-FMPP has a possibility to enhance the current Von Neumann computing system considerably. To enjoy such enhancement as much as possible, huge computation space should be prepared. We have to consider more dense and powerful PE structure.

The developed BPBP-FMPP LSI have various functionalities. The rich functionalities and 32bit word structure diminishes the number of PEs on a single LSI, which results that only eight PEs are available. The 32bit structure also decreases the processing speed. As shown in Figure 4.11, peripheral circuitry besides the memory block occupies half of the PE area. To enhance the integration density, the bit width of words and an ALU should be decreased to be optimized for some specific applications. The peripheral circuitry is also minimized. In the next chapter, we explain an application specific FMPP called FMPP-VQ. The FMPP-VQ64 LSI integrates 64 PEs. A PE consists of 16word 8bit SRAMs and an 12bit ALU.

Chapter 5

Functional Memory Type Parallel Processor for Vector Quantization: FMPP-VQ

An application specific FMPP called "FMPP-VQ" is discussed in this chapter. The FMPP-VQ accelerates the nearest neighbor search on vector quantization (VQ). It can be applied to low-rate video compression. Three LSIs are already available: FMPP-VQ4, FMPP-VQ64 and FMPP-VQ64M. The FMPP-VQ4 integrates 4 PEs to evaluate its functionality. The latter two LSIs integrate 64 PEs, which can be applied to actual low-rate video compression. The FMPP-VQ64 and FMPP-VQ64M achieve both of high performance and low power. We have also developed a low-rate video compression system and a multi-stage hierarchical vector quantization algorithm using the FMPP-VQ.

5.1 Introduction

We can currently use cellular phones and PHS[1] for the mobile sound communication. Visual information is important to communicate each other because we can easily recognize and analyze information with our eyes. In the mobile communication, transmission of information is restricted within a limited bandwidth. Visual information, however, involves a huge amount of data. For example, a frame of full-color images including 176×144 pixels amounts to 76kbytes. The PHS which achieve a wide bandwidth of 32kbps cannot send even a single frame per second without compression. Thus, some kind of data compression technique must be applied. In addition, the data compression technique should also be low-power consuming, since all of electric instruments for mobile computing are driven by batteries. In the JPEG or MPEG algorithms the discrete cosine transformation (DCT) compresses space redundancy of an image. Such DCT-based image compression algorithms achieve both high quality and high compression ratios, but consume large amount of hardware and power both during compression and during decompression. On the other hand, vector quantization (VQ)[GC83] is a promising candidate for low-rate and low-power image compression, since it requires much less hardware for decompression. During compression, however, it requires a large amount of

[1] Personal Handy-phone Systems available in Japan.

computation time. The most time-consuming factor in compression is the "nearest neighbor search," which searches for a vector nearest to an input among several vectors.

The memory-based SIMD shared-bus parallel processor architecture, the FMPP is well suited for computing the nearest neighbor search. We propose an implementation of the FMPP architecture to accelerate the nearest neighbor search.

The developed hardware is called "FMPP-VQ", to signify an FMPP for Vector Quantization[KKT+96]. It has as many processing elements (PEs) as code vectors. A shared bus connects all PEs. The nearest vector is searched exhaustively in parallel. Each PE has conventional memories to store code vectors and an arithmetic logic unit (ALU) based on pass-transistor logic to compute the distance between an input vector and code vectors. The nearest vector is obtained using the CAM-based parallel search. These procedures are done in O(k), where k stands for the dimension of vectors. The number of code vectors does not affect computation time. In the nearest neighbor search, only input vectors are broadcast to PEs from a shared data bus. The distance is locally computed in each PE. Thus, memory-based PEs perform effective computation to obtain an input vector broadcast through a shared bus. Code vectors can be easily updated, since they are stored in conventional memories. All PEs can be laid out in a memory-like regular-array structure, which minimizes circuit area. A large number of PEs can be integrated on a single LSI and used to perform massively parallel computation. We have designed and fabricated three LSIs of the FMPP-VQ architecture. The first LSI called "FMPP-VQ4" was fabricated in 1996, which integrates four PEs to evaluate functionality of the FMPP-VQ. It is almost fully functional. Then we designed and fabricated the FMPP-VQ64 integrated 64 PEs in 1997. PE arrays are fully-functional, but its control logic has some design errors. These two LSIs work at 25MHz clock cycles. The power dissipation of the FMPP-VQ64 is 20mW at 25MHz clock frequency and 3.0V power supply. It performs 53,000 nearest neighbor searches per second. It can be applied to the image compression on the mobile computing field. The third LSI called FMPP-VQ64M is developed to achieve higher performance and lower power. Its performance is doubled, while its power dissipation is half compared with the FMPP-VQ64.

We develop a video compression system using the FMPP-VQ64. It consists of the FMPP-VQ64 LSI and an FPGA for control logic, attached to personal computers. Images compressed by vector quantization can be easily decompressed. A serial commercial processor for PDA has enough power to decompress the compressed images in real time. The developed compression algorithm can compress a frame of a QCIF (176$\times$144) video sequence into 2920bits. The quality of compressed images is over 30dB for some standard video sequences like "Suzie" or "Miss America." We also develop an evaluation encoding system consists of a host computer and an FMPP-VQ64 LSI. It compresses 10 QCIF frames per second in real time. A PHS terminal can send 10 frames per second

via the 29.2kbps mobile wireless channel. The algorithm is very robust to noise, since indexes from the nearest neighbor search is coded with a fixed length.

In this chapter, Section 5.2 gives a brief introduction of vector quantization. Section 5.3 shows how to accelerate the nearest neighbor search on the FMPP architecture. The architecture and structure of the FMPP-VQ is explained in Section 5.4 The detail description of the FMPP-VQ4 and FMPP-VQ64 are shown in Section 5.5. The FMPP-VQ64M are described in Section 5.6. The FMPP-VQ64M is now under test. We compare the FMPP-VQ with some other vector quantizer and commercial sequential processors in Section 5.7. Section 5.8 gives a description of the compression algorithm and the experimental real-time low-rate video compression system.

5.2 Vector Quantization of Image

Vector quantization (VQ) can be defined as a form of pattern recognition where an input pattern is "approximated" by one of a set of patterns[GG92]. It is mostly applied to image compression. An input image is first divided into meshes which includes $\sqrt{k} \times \sqrt{k}$ pixels. Each mesh is then approximated by one of a set of patterns. We call a mesh an input vector $\vec{x}$ and a set of patterns a codebook Y. A vector in a codebook is referred to as a code vector $\vec{y_i}$. For VQ to be put into practical use, there are two issues that need to be addressed. One is the acceleration of the "nearest neighbor search (NNS)." The NNS searches for a vector nearest to an input vector among a large number of vectors. This requires a substantial amount of time on conventional serial processors in relation to the dimension and the number of code vectors. The other issue is the design of an optimal codebook. The quality of reconstructed images obtained from a common set of images using DCT-based compression algorithms is independent of the algorithm used. In VQ, however, the quality of the compressed image depends on the codebook design. The proposed hardware, FMPP-VQ is designed to satisfy these two considerations, acceleration of nearest neighbor search and optimization of codebooks.

We should perform optimization sequence to find an optimal codebook. For example, the LBG[LBG80] algorithm is one of the most famous algorithms to find an optimal codebook. It usually requires long training sequence to obtain an optimized codebook.

Figure 5.2 explain the nearest neighbor search and codebook optimization for two-dimensional vectors. In the LBG algorithm, a code vector is rearranged into the center of gravity among all the nearest vectors.

Table 5.1 shows parameters and definition for vector quantization.

The nearest neighbor search (NNS) can be defined as Equation (5.1).

$$i_{nearest} = \min_i{}^{-1} d(\vec{x}, \vec{y_i}) \tag{5.1}$$

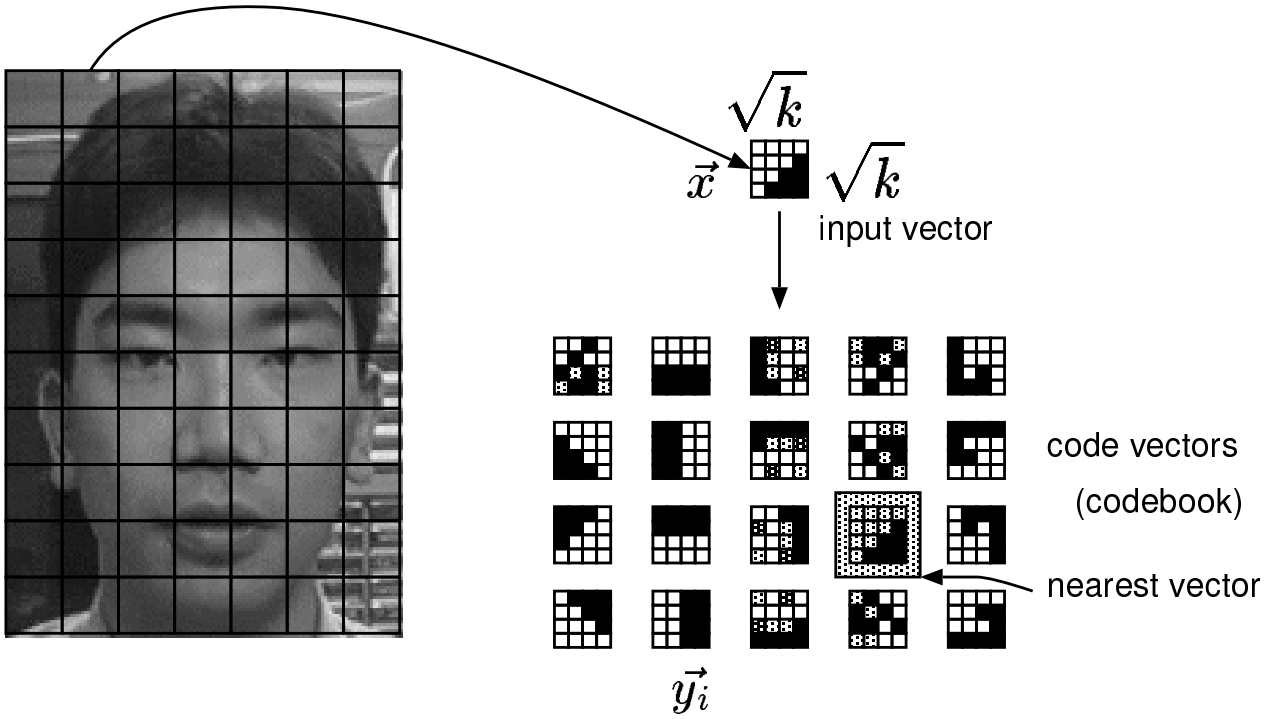


Figure 5.1: Vector quantization of images.

$d(\vec{x}, \vec{y_i})$: distance between $\vec{x} and \vec{y_i}$

To compute the distance between vectors, two measurement methods are well known. One is the absolute distance (mean absolute error (MAE)) in Equation (5.2).

$$d(\vec{x}, \vec{y_i}) = \sum_{j=0}^{k-1} |x_j - y_{ij}| \tag{5.2}$$

The other is the squared distance (mean squared error (MSE)) in Equation (5.3).

$$d(\vec{x}, \vec{y_i}) = \sum_{j=0}^{k-1} (x_j - y_{ij})^2 \tag{5.3}$$

The squared distance requires multiplication to compute the second power products, which wastes the silicon area. On the other hand, the absolute distance requires no multiplication, but it may fall into a local optimum solution. Code vectors for images, however, are spread out sparsely in the Euclidean space. The absolute distance has enough quality for image compression.

The FMPP-VQ performs the nearest neighbor search to simultaneously compute all the absolute distances between a broadcast input vector and code vectors. The memory-based architecture of the FMPP enables optimization of the code vectors, since they are stored in memory cells in the FMPP-VQ. It can be accessed as the same manner than conventional memory devices.

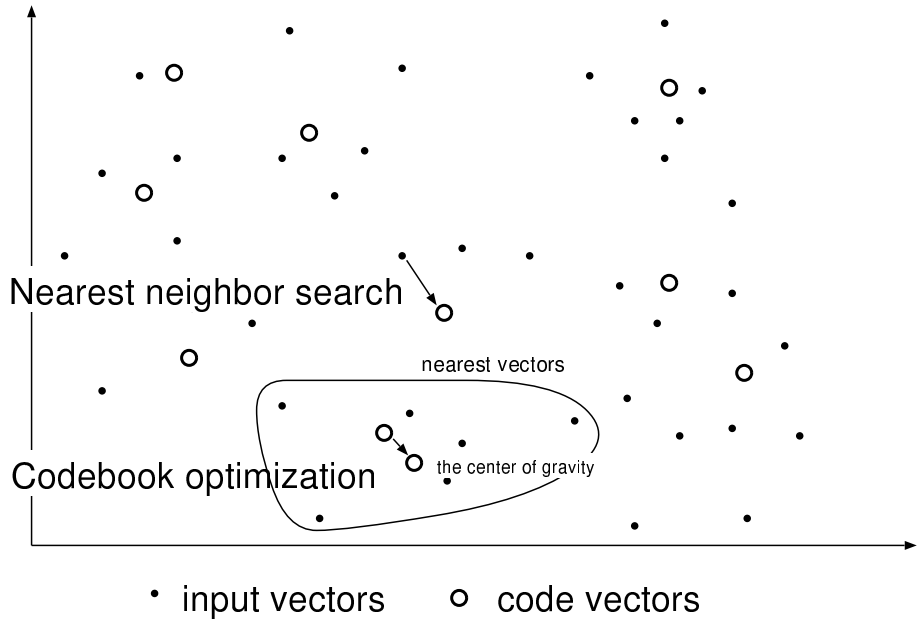


Figure 5.2: Nearest neighbor search and codebook optimization.

5.3 Vector Quantization on the FMPP

The first FMPP LSI called the BPBP-FMPP described in Chapter 4 appeared in 1994[KOT95]. It involves 8 processing elements. Each processing element (PE) consists of several words of CAM (Content Addressable Memory) and an ALU. The ALU can perform addition and logical operations, which are implemented using a CAM-based pass transistor logic. The BPBP-FMPP can search for the minimum value among all words.

The NNS of vector quantization can be an application very suitable for the FMPP. It requires an input vector and a set of code vectors. In order to compute the NNS in the FMPP, PEs store code vectors and an input vector is broadcast to all PEs through a bus. First all distances between the input vector and the code vectors are computed. The code vector nearest to the input is then extracted. PEs can compute the distance locally without communication between PEs. The absolute distance $\Sigma|\vec{x}-\vec{y}|$ can be calculated with a combination of subtraction and logical operations already implemented in the BPBP-FMPP. The minimum absolute distance can be extracted using its minimum value search capability. Thus, the FMPP architecture effectively accelerates the NNS. Code vectors can also be easily optimized since they are stored in a conventional memory (CAM) and can be read or written in the same manner with a conventional memory.

We have designed an LSI called the FMPP-VQ to be optimized for the NNS. To confirm the functionality of the FMPP-VQ, we have developed and fabricated the FMPP-VQ4 which contains four PEs[KKT$^+$97]. It operates properly at 25MHz. Four PEs are enough to verify parallel SIMD

Table 5.1: Parameters and definition for vector quantization.

Parameter/Definition	Synopsis
k	dimension of vectors
N	number of code vectors
m	bit width of vectors
Y	codebook
$\vec{y_i}$	ith code vector in Y
y_{ij}	jth element of ith code vector
$\vec{x}$	input vector
x_j	j th element of input vector

operations, but insufficient for actual image compression. We have developed and fabricated the FMPP-VQ64[KNT+98] with 64 PEs for real-time low-rate image compression. The FMPP-VQ64 is fully-functional and performs 53,000 NNSs per second, while its power consumption is 20mW at 25MHz clock frequency and 3.0V power supply. We have also developed an image compression algorithm using vector quantization for the FMPP-VQ. A low-bit rate image compression system is now under development, sending 10 frames of QCIF (176×144) video sequence through a 29.2 kbps wireless line of the PHS. The image compression system consists of a single LSI of the FMPP-VQ64, an FPGA for control logic and a host computer. The FMPP-VQ achieves both of high performance and low power. But the simple control logic does not work because of some design faults. We have designed and fabricated an modified versions of the FMPP-VQ64 called FMPP-VQ64M. It contains more sophisticated control unit to manage the nearest neighbor search. Its throughput is increased to 91,000 NNSs per second, while its power consumption is estimated to be 10mW.

5.4 Architecture and Structure

The features of the FMPP-VQ are summarized as follows.

- The bit-parallel block-parallel structure is adopted.
- The absolute distance $\Sigma|\vec{x} - \vec{y}|$ is used as the distance measure.
- The nearest neighbor search is computed in O(k). (k denotes the vector dimension.)
- 16 dimensional vectors are used.

- Code vectors are stored in SRAM cells.
- A pass-transistor based arithmetic logic unit (ALU) computes the absolute distance between $\vec{x}$ and $\vec{y_i}$
- The minimum distance is extracted using the CAM-based search procedure.

The distances between a broadcast input vector and code vectors are computed on all the PEs in an SIMD manner. We can get the nearest neighbor vector by finding the minimum value from all the distances. Since conventional adders consume hardware, we use an adder based on a pass-transistor logic.

5.4.1 Nearest Neighbor Search on the FMPP-VQ

Required operations for the nearest neighbor search are listed as follows.

1. Absolute distance measurement. $|\vec{x} - \vec{y_i}|$
   - 1.1 Subtraction. $x_j - y_{ij}$
   - 1.2 Absolute value computation(ABS). $|x_j - y_{ij}|$
   - 1.3 Accumulation. $\sum_{j=0}^{k-1} |x_j - y_{ij}|$
2. Minimum value search. $\min_i |\vec{x} - \vec{y_i}|$

The FMPP-VQ is designed to perform these operations. Absolute distances are computed on all the memory-based PEs, and then the minimum value is searched in parallel. The size of codebooks does not affect the time to compute the NNS, since all the distances are computed in parallel and we use the minimum search procedure similar to that on conventional CAMs [OYN85, OYY86].

In order to compute the absolute distance, the capability of numerical operations are implemented in the PE. In the FMPP-VQ, numerical operations are done using the same strategies as the BPBP-FMPP according to Equations (5.4)-(5.7).

$$C_i = G_i + P_i \cdot C_{i-1} \tag{5.4}$$

$$G_i = A_i \cdot B_i \tag{5.5}$$

$$P_i = A_i | B_i \tag{5.6}$$

$$S_i = \overline{C_{i-1} \oplus \overline{P_i}} \tag{5.7}$$

In the BPBP-FMPP, addition is composed of several individual operations: logical operations and carry propagation. It takes 6 steps to complete addition. On the other hand, numerical operations on the FMPP-VQ can complete in a single cycle. The carry propagate (P) and the carry generate (G) are produced simultaneously. Then, they are sent to the carry chain. Finally the XNOR operation between the carry and P is done. There is no buffer or word to save P, G and C. The required number of transistors can be reduced compared with that of the BPBP-FMPP.

5.4.2 Structure of the FMPP-VQ

Figure 5.3 shows a block diagram of the FMPP-VQ. Each PE stores a code vector $\vec{y}_i$ in a codebook Y and computes $|\vec{x} - \vec{y}_i|$. All PEs are laid out in a $\sqrt{N} \times \sqrt{N}$ two-dimensional regular array structure and connected through a shared bus called the "global data bus." An input vector is broadcast element by element through the global data bus. In the PE, the absolute distance of each element $|x_j - y_{ij}|$ is accumulated element by element. After all elements are broadcast, the minimum value is extracted as the same manner as the conventional CAMs.

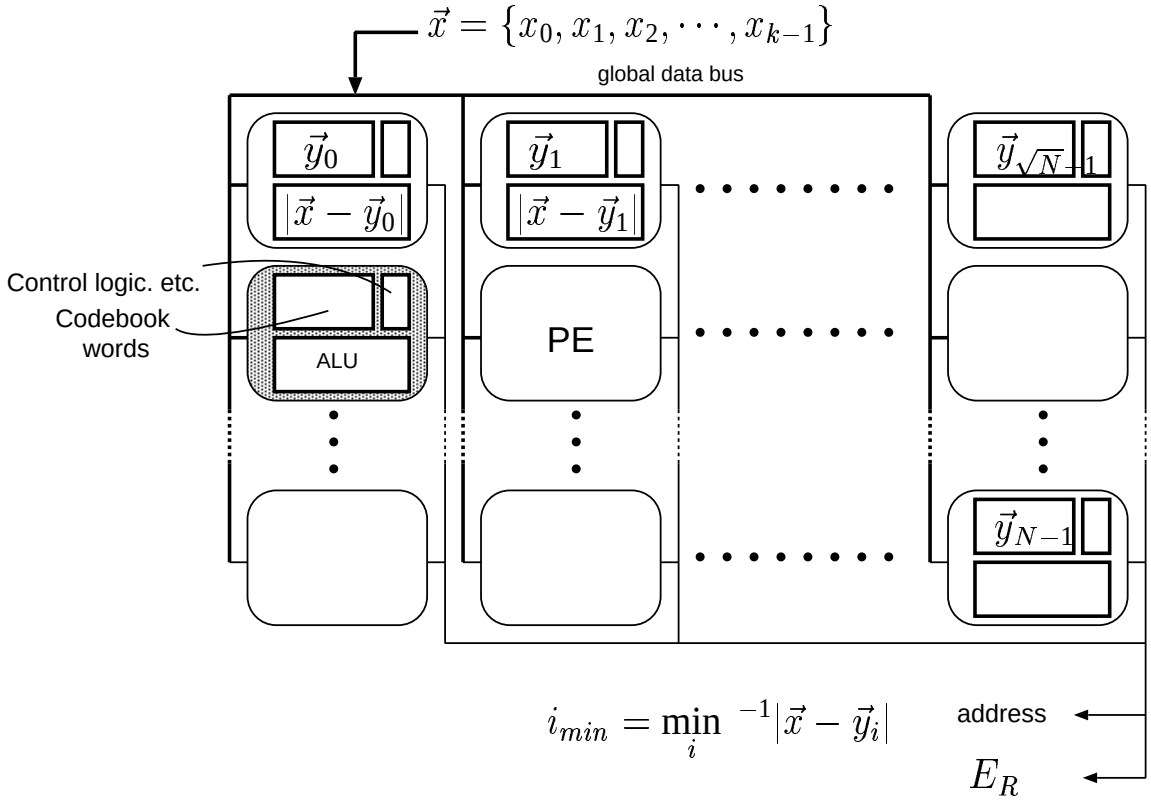


Figure 5.3: Block diagram of the FMPP-VQ.

The PE contains 16 codebook words for a code vector and an ALU to compute the absolute distance. To accumulate the absolute distance of the lth element to the $(l-1)$ accumulated result,

Equation (5.8) is performed. Since the FMPP-VQ is an SIMD parallel processor, all PEs must perform the same operations. To compute the absolute distance $|x_j - y_{ij}|$, operations should be given individually to each PE according to the values of x_j and y_{ij}. There are two ways to compute the absolute value, depending on whether an element of a code vector is greater than (Condition 1) or less than (Condition 2) an element of an input vector.

$$\sum_{j=0}^{l} |x_j - y_{ij}| = \begin{cases} \sum_{j=0}^{l-1} |x_j - y_{ij}| + y_l - x_l & (y_l \geq x_l\text{: Condition1}) \\ \sum_{j=0}^{l-1} |x_j - y_{ij}| + \overline{y_l - x_l} + 1 & (y_l < x_l\text{: Condition2}) \end{cases} \tag{5.8}$$

The FMPP-VQ computes the absolute distance to repeat Equation (5.8) from $l = 0$ to $k - 1$. To compute the NNS of these two conditions on an SIMD processor, we divide Equation (5.8) into the following three operations.

Operation 1 Compute $y_l - x_l$.

Operation 2
- Condition 1 $[(y_l - x_l \geq 0)]$ Nothing done.
- Condition 2 $[(y_l - x_l < 0)]$ Compute $\overline{y_l - x_l}$.

Operation 3
- Condition 1 Accumulate $y_l - x_l$ to $\sum_{j=0}^{l-1} |x_j - y_{ij}|$.
- Condition 2 Accumulate $(\overline{y_l - x_l} + 1)$ to $\sum_{j=0}^{l-1} |x_j - y_{ij}|$.

The overflow bit of the carry is memorized at the subtraction of Operation 1. It is used to control the following two operations.

Figure 5.4 shows the structure of the PE. A codebook word CW(j) stores y_{ij} in a code vector $\vec{y_i}$. For vector quantization of image, the dimension of vector k is usually 16. Thus, each PE of the FMPP-VQ has 16 codebook words. A memory cell of the codebook word is a conventional 6-transistor SRAM. The operand word OW stores an operand on every operation. The result word RW stores $\sum |x_j - y_{ij}|$. The temporary word TW stores results from operations. The local data bus (lb0, lb1) connects operand words and the other words in the ALU. There is no conventional adder in the ALU. Instead, the operand word, carry chain and the XNOR gate work together for addition or subtraction according to Equation (5.9), which can be obtained from Equations (5.4)-(5.7).

$$A_i + B_i = \overline{(A_i \oplus B_i) + \overline{C_{i-1}}} \tag{5.9}$$

The operand word is designed using the conventional SRAM-based CAM[OYN85]. Logical operations required for addition are executed using pass transistors in the operand word. The carry chain is also composed of pass transistors, which accelerates carry propagation. Thus, addition

can be done in bit-parallel. A codebook word consists of 8bit-wide SRAMs, while the ALU is 12bit-wide because the distance between an input vector and a code vector may grow as wide as 12 ($= m + \lfloor \log_2 k \rfloor = 8 + \lfloor \log_2 16 \rfloor$)bit. The overflow flag OF stores the overflow bit from the carry chain. The OF is connected to the local control logic (LCL), which controls several input signals in the PE for Operation 2 and 3. The search flag SF is used to store the result from the search operation. On the search operation, a search key is given from the global data bus. Then, it is compared with the value stored in the OW . If the search key is equal to it, the SF becomes true. The signal C_R from the SF is linked to the local priority address encoder. Its detailed description is given in the following section.

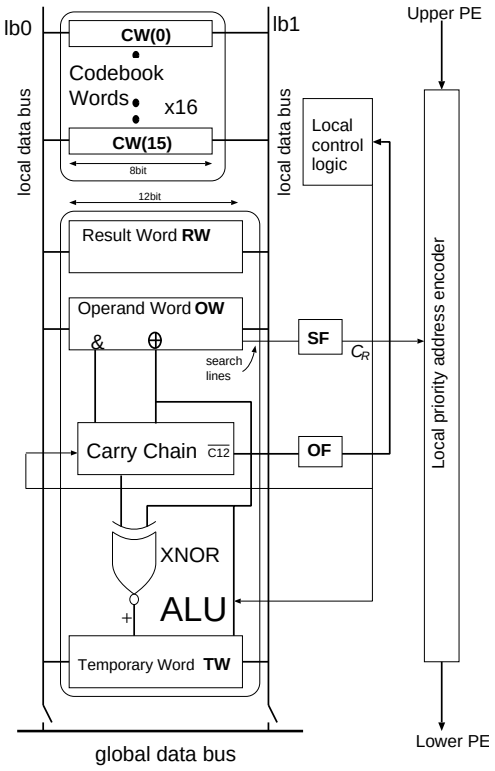


Figure 5.4: Structure of a PE.

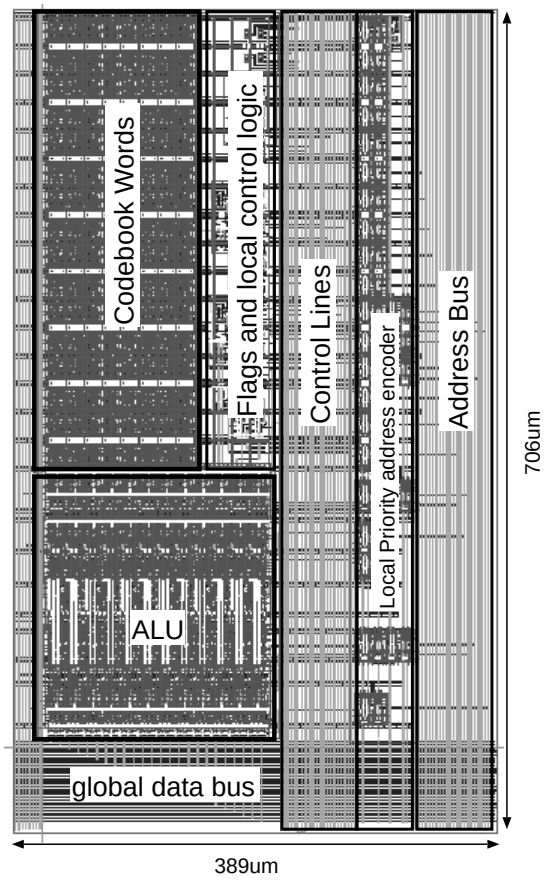


Figure 5.5: Layout pattern of a PE.

5.4.3 Detailed Structure of the PE

The block diagram of the PE is already shown in Figure 5.4. Here, transistor level structures of the PE is described in detail.

As shown in Figure 5.4, the PE of the FMPP-VQ consists of operand words, the ALU, the flags and the local control logic. All of these components are laid out in a square region to satisfy the

two-dimensional regular structure. Since the operand words is 8-bit wide and the ALU is 12-bit wide, the flags and the local control logic are laid out in the empty space at the side of the operand words. The layout pattern of the PE is shown in Figure 5.5. It is implemented in a rectangle region to satisfy the two-dimensional regular array structure. In the left side of the PE, there is the local priority address encoder to search for the nearest vector.

Operand words and the ALU

The memory cell of the operand word OW is a conventional 6 transistor SRAMs already shown in Figure 2.4. Figure 5.6 shows one bit slice of the ALU. All three words, the result word, the operand word and the temporary word consist of an 8 transistor SRAM cells. Between the operand word and the temporary word, the carry chain and the XNOR gate are placed.

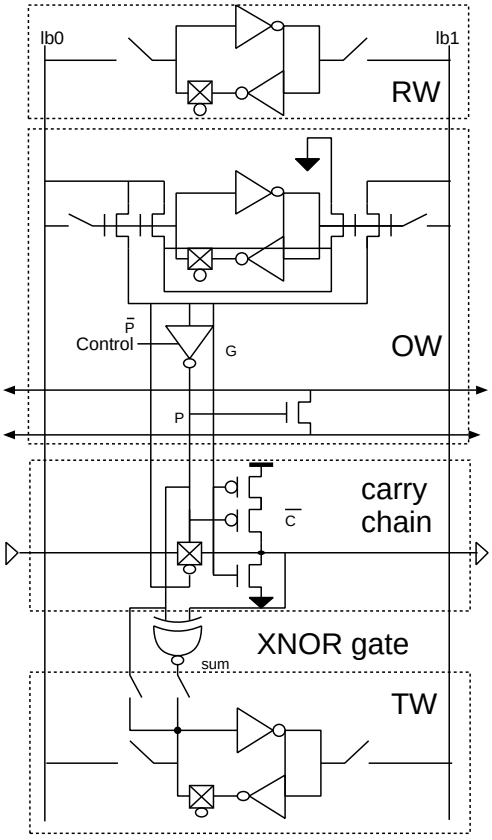


Figure 5.6: One bit slice of the ALU.

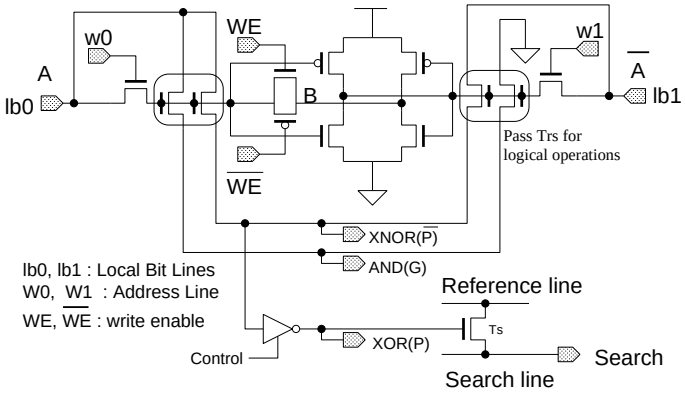


Figure 5.7: Operand word.

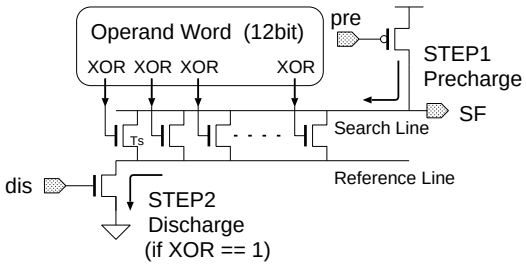


Figure 5.8: Search line and reference line for the search operation.

Figure 5.7 shows a memory cell of the operand word OW. It stores an operand in the SRAM cell and produces logical AND (G), XOR (P) and XNOR ($\overline{P}$) values between a stored value and a broadcast value through the local bit lines. The four pass transistors create logical AND and XNOR. The transistor Ts is used for the search operation. Figure 5.8 shows the search line and the reference line for a single 12-bit operand word. The search line is precharged before the search operation. When all the XOR values are fixed according to a broadcast key data, the reference line is discharged.

If the key data is equal to the value in the operand word , all the XOR values becomes false. Thus, the search line remains high voltage. If not, it is discharged. This capability is almost same as that of conventional CAMs. The search line is connected to the search flag. Figure 5.9 shows two bit slice of the carry chain. It produces the carry $\overline{C}$ from the carry propagate P and the carry generate G. The input node P and $\overline{P}$ and G are connected to the equivalent output nodes of the OW. The functionality of the carry chain is entirely same as that in the BPBP-FMPP. In the BPBP-FMPP, the carry chain is activated by the clock signal, while there is no clock signal in Figure 5.9. Its area is decreased, but it may be activated whenever the states of input signals are changed. In the FMPP-VQ64, the inverter in the OW is activated by a control signal in order to eliminate unnecessary state changes of the input signal P. Figure 5.10 shows the schematic diagram of the inverter controlled by an NMOS FET.

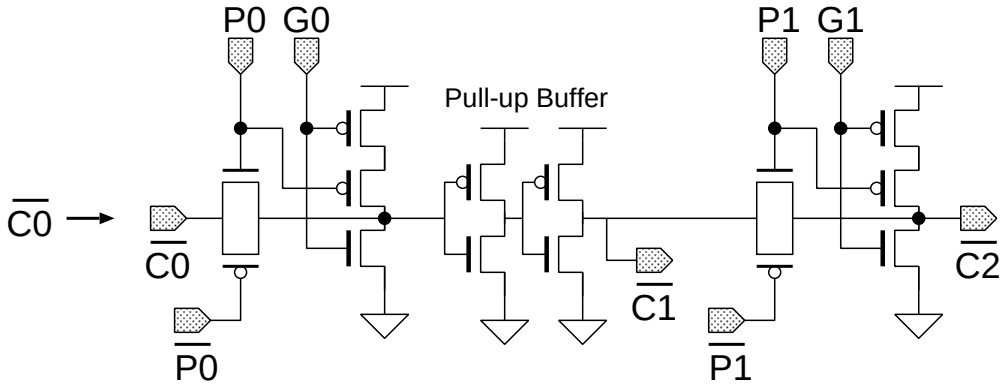


Figure 5.9: Two bit slice of the carry chain.

Figure 5.11 shows the XNOR (exclusive-nor) gate which produces the sum from the carry propagate and the carry. It consists of 6 transistors.

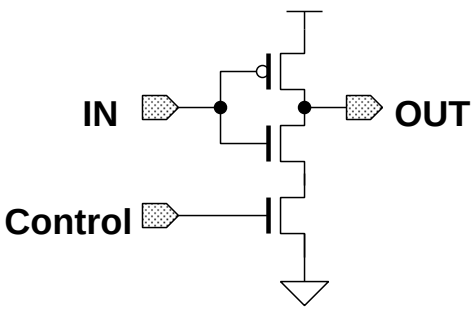


Figure 5.10: Inverter controlled by an NMOS FET.

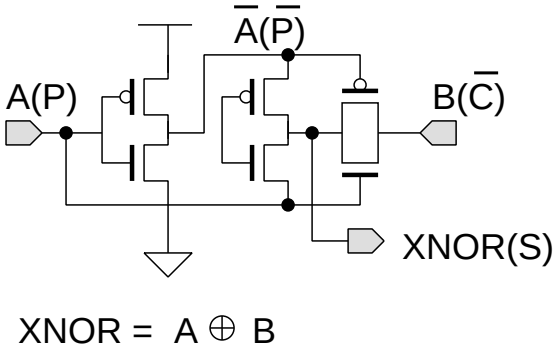


Figure 5.11: XNOR (exclusive-nor) gate.

Figure 5.12 shows the temporary word TW. The input node P is connected to the correspond output node of the OW. The control node Pin is activated at the complement operation in the absolute value computation. The input node S is connected to the corresponding node of the XNOR

gate.

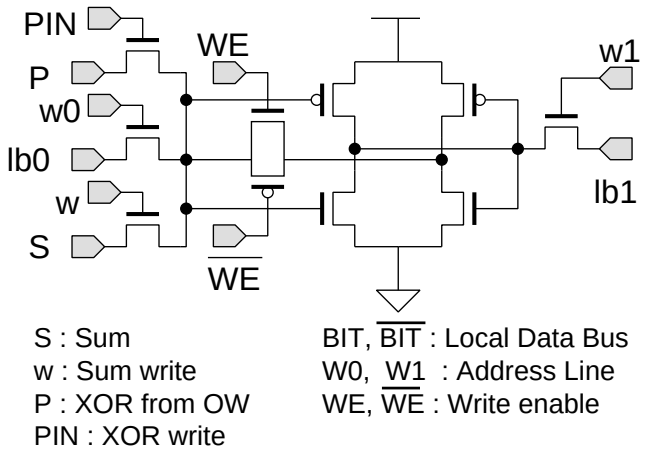


Figure 5.12: Temporary word.

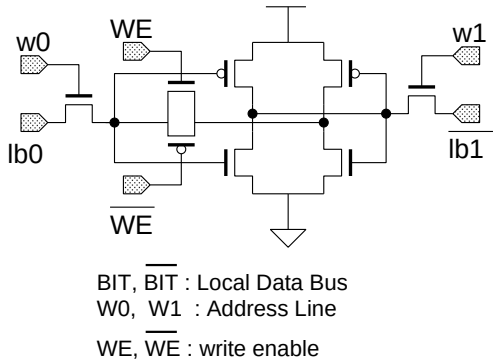


Figure 5.13: Result word.

Figure 5.13 shows the result word RW. It is an 8-transistor SRAM cell which has a CMOS pass-transistor switch.

The operand words consists of the 6-transistor SRAM cell, while the other words has an additional CMOS switches to cut off the inverter loop in the SRAM cell. This is because the operand word are always modified by the driver outside of the PE, while the other words may be modified by the internal words in the PE. The CMOS switch must be off on the word that the other internal word is going to write.

Two flags and the local control logic

Figure 5.14 shows the schematic diagram of two flags, the search flag SF and the overflow flag OF. The input node IN of the SF is connected to the search line in the operand word. That of the OF is connected to the MSB of the carry in the carry chain. These two flags are also connected to the local data bus for test. Their values can be read or written directly from the data bus. The details is written in Section 5.5.4.

Figure 5.15 shows the OF and the part of the local control logic connected to it. The OF stores the overflow at the subtraction on Operation 1 in page 59. Then the local control logic produces the signals PIN and $\overline{C0}$ according the the value of the OF at Operation 2 and 3. The other part of the local control logic defines the operation mode of the PE whether it operates by SIMD operations or by block-oriented operations, which is explained later in Section 5.5.4.

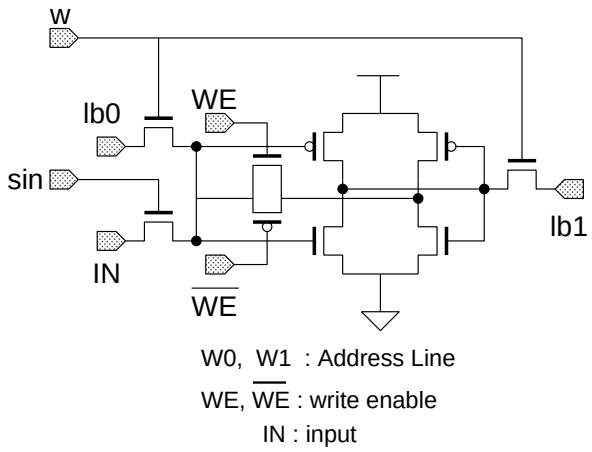


Figure 5.14: Schematic view of two flags.

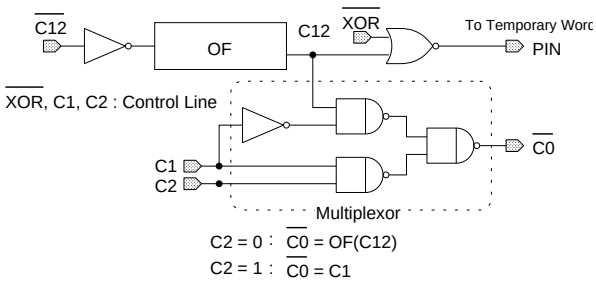


Figure 5.15: The overflow flag and the part of the local control logic.

Priority address encoder

The priority address encoder is used to extract the minimum value. It consists of comparison lines, a priority encoder and an address encoder. The comparison lines rapidly confirm whether there is any true search flags in the FMPP-VQ. The priority encoder resolves the word which contains the minimum value. Figure 5.16 shows the column priority address encoder for 8 PEs. It is for 64PEs of the FMPP-VQ64. The precharged comparison lines are rapidly discharged if there is any true search flag. It is done as the same manner with the search and reference lines in the OW. The priority address encoder consists of a priority encoder and an address encoder. It resolves the topmost word among all the words whose SFs are true and then outputs its address. The local priority address encoder is a single PE slice of the priority address encoder.

Figure 5.17 shows the structure of the two dimensional priority address encoder. The row priority address encoder has the same structure with the column one. Comparison lines in the row and column priority address encoder work to output E_R which becomes false if there is no true search flag. The priority encoder in the row priority address encoder resolves the lowest row and the address encoder outputs the row address. The column address of the resolved row is selected by the AND gates and wired-OR logic placed below the row priority address encoder.

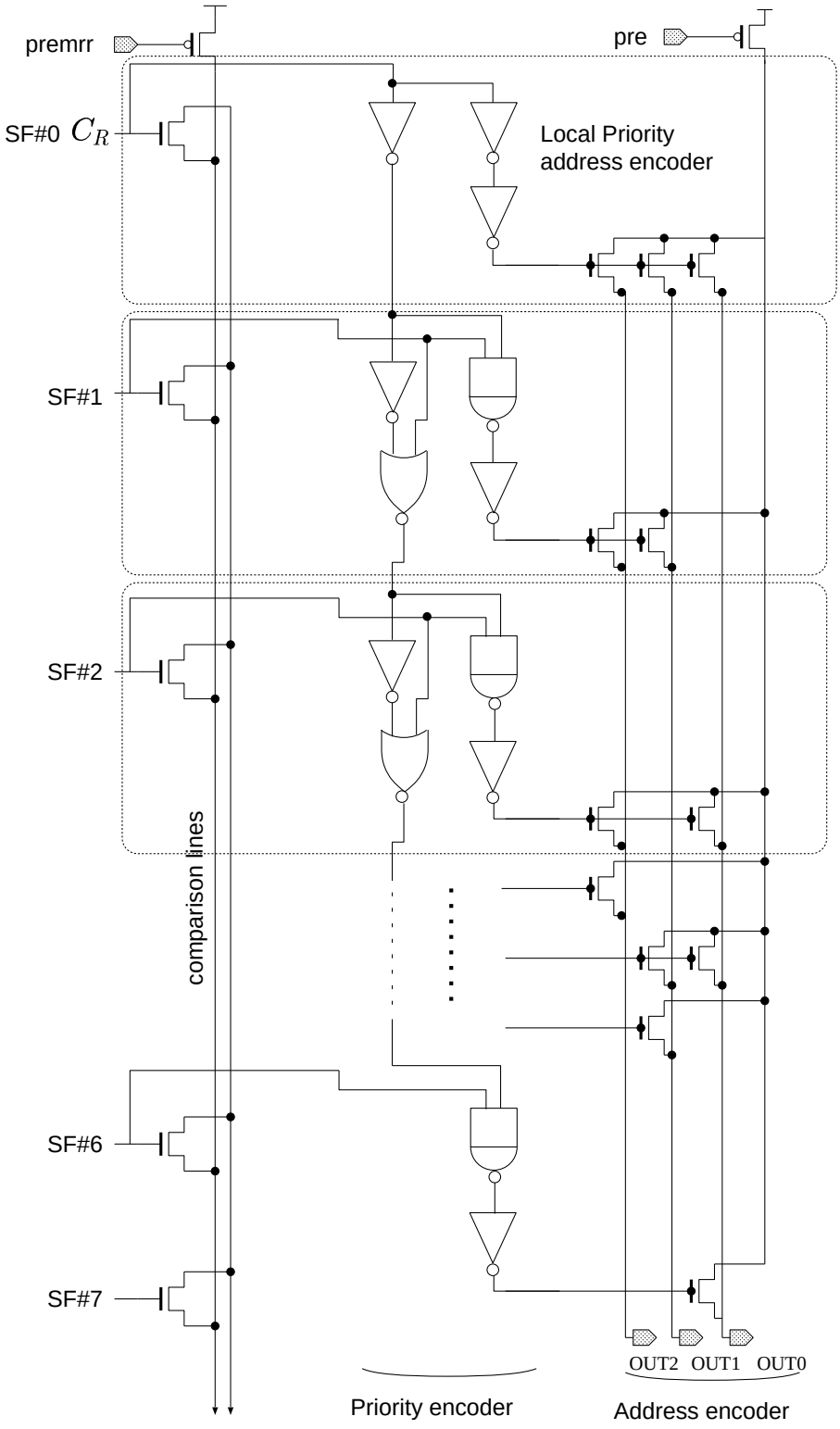


Figure 5.16: Column priority address encoder.

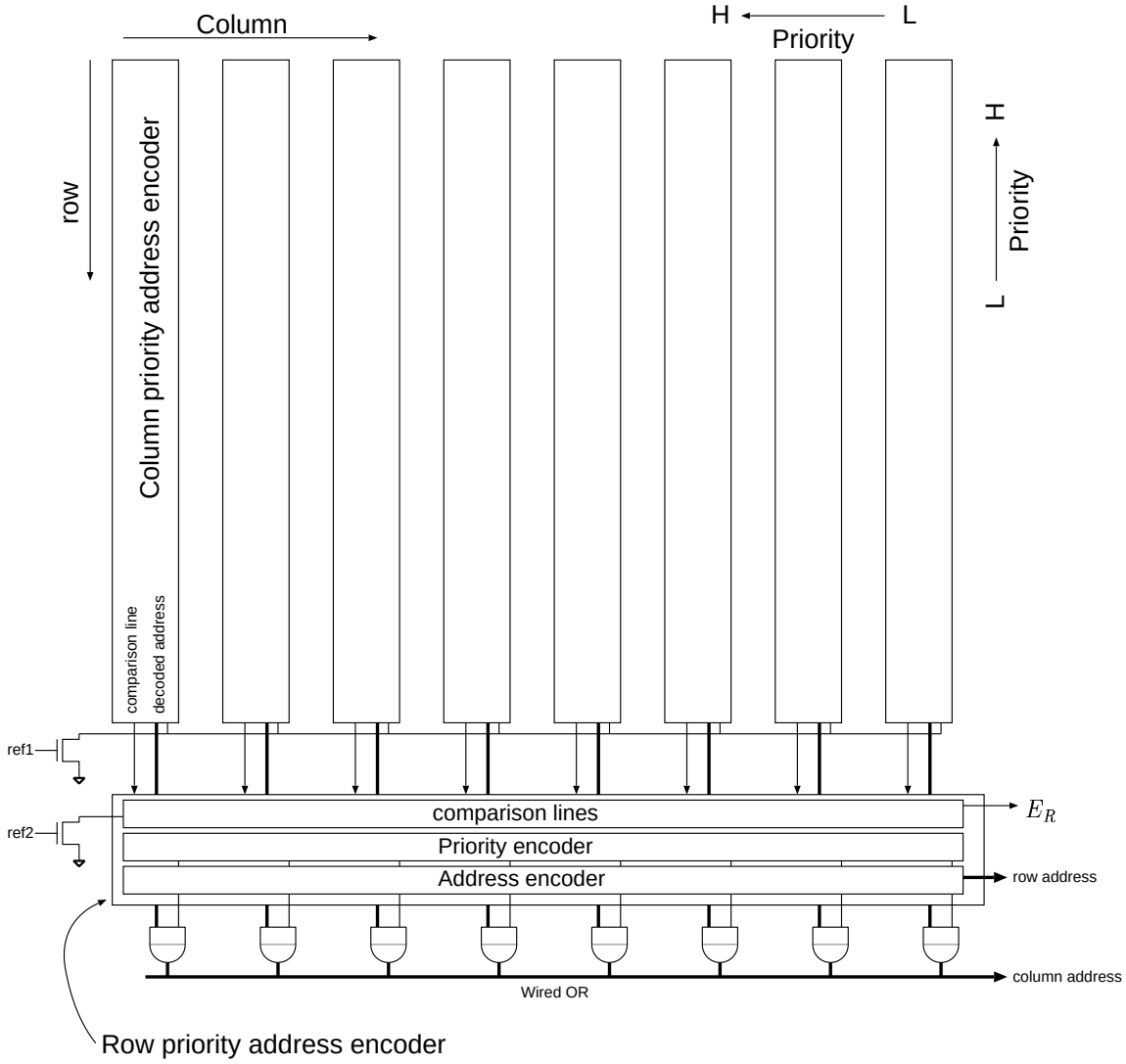


Figure 5.17: Two dimensional priority address encoder for the FMPP-VQ64.

5.4.4 Nearest Neighbor Search Procedure

The nearest neighbor search consists of the absolute value computation and the minimum value search. Here, the absolute distance computation and minimum value search method is described in detail using the data flow in the PE.

The absolute residual value of single elements of $\vec{x}$ and $\vec{y}_i$ is accumulated to the last absolute distance according to Equation (5.8) (in page 59). Each element x_j of $\vec{x}$ is given one by one to every PE through the global data bus. Every time an element x_j of $\vec{x}$ comes to a PE, $\sum |x_j - y_{ij}|$ is computed according to the following procedure.

Step1: [code2operand(j)**]**	The jth element of $\vec{y}_i$, y_{ij} is sent to the OW.
Step2: [subtraction(x_j)**]**	The inverse value of x_j, $\overline{x_j}$ is broadcast to all PEs. Then, the subtraction, $y_{ij} - x_j$ is done. The result is stored in the TW. The overflow value is stored in the OF.
Step3: [temp2operand**]**	The value stored in the TW is transfered to the OW.
Step4: [complement**]**	The TW becomes the complement value of the OW. The PEs where $(y_{ij} - x_j) < 0$ perform the operation.
Step5: [temp2operand**]**	Same as **Step 3**.
Step6: [addinner**]**	Addition between $\sum_{l=0}^{j-1} \|x_l - y_{il}\|$ and the OW is done. if $(y_{ij} - x_j) < 0$, the 0th bit of the carry is set to 1 at the addition.
Step7: [temp2result**]**	The value stored in the TW is transfered to the RW . RW becomes $\sum_{l=0}^{j} \|x_l - y_{il}\|$

It consists of 7 steps (operations). Four operations code2operand, temp2operand$\times$2, temp2result transfer values from any word to any. There are 3 operations subtraction, complement, addinner between these transfer operations, which correspond to Operation 1, 2 and 3 in page 59 respectively. Figure 5.18 shows the flow of a single-dimension slice of the absolute distance computation. The upper PE represents Condition 1 and the lower one represents Condition 2. The OF stores an overflow value on the subtraction at **Step 2**. The OF and the local control logic in Figure 5.15 disable the complement at **Step 4** and determines the carry on the addinner at **Step 6**. Note that the subtraction adds the 2's complement value of x_i (=5) and a value stored in the OW. The absolute distance can be computed to repeat the flow 16 (=k) times.

After all elements of $\vec{x}$ are given, the CAM-based parallel search procedure finds the minimum value. The output signal E_R from the comparison line in Figure 5.17 becomes false if a broadcast search key is not matched to any operand word. Figure 5.19 shows a program of the minimum value search. The definitions of these operations are given later in Table 5.2 and Table 5.3. Figure 5.20 is the procedure for the minimum value search among four 5bit values. The minimum distance can

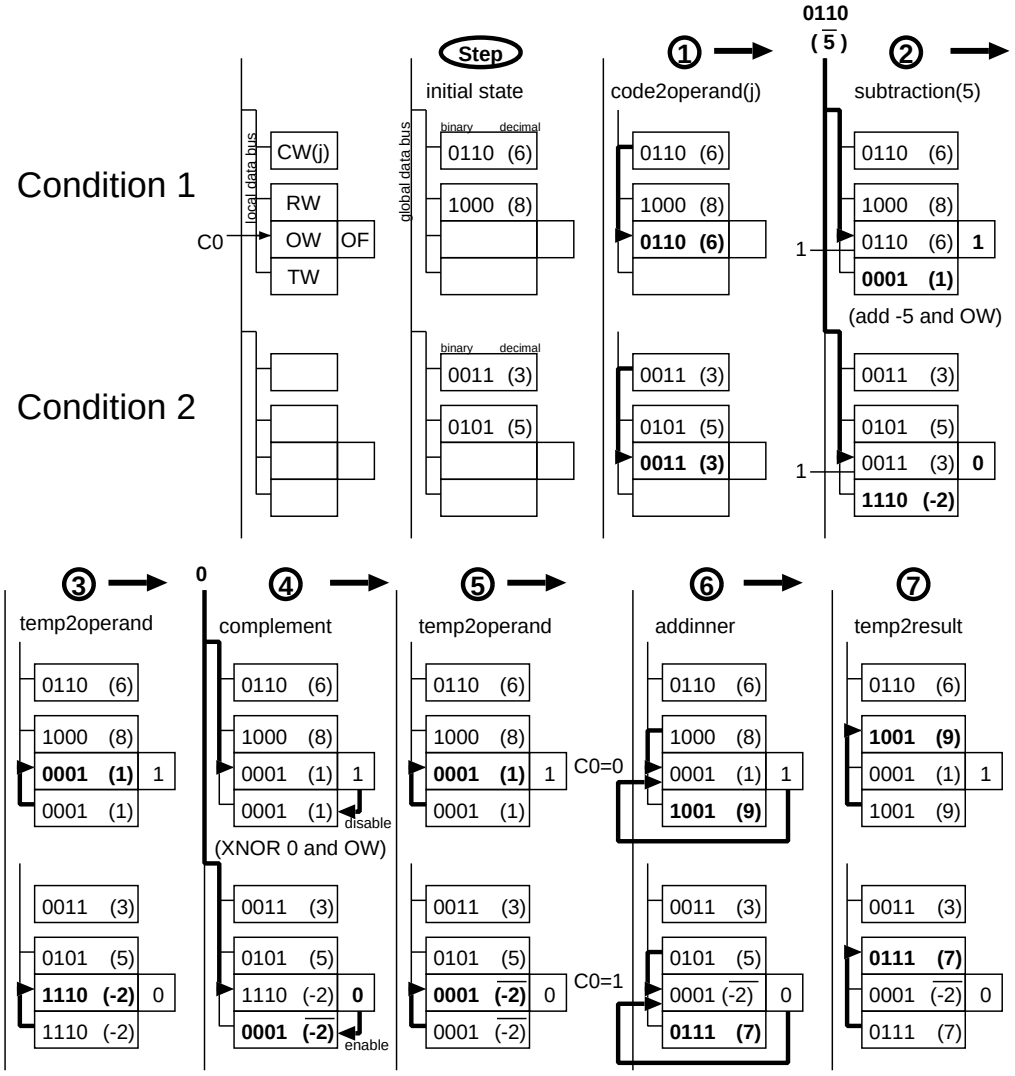


Figure 5.18: Procedure for computing the absolute distance.

be extracted to repeat a set of the search and mrr operations from MSB to LSB 12 ($= m + \lfloor \log_2 k \rfloor$) times. The place which contains the minimum value is extracted by the priority address encoder.

5.4.5 List of Operations on the FMPP-VQ

All possible operations of the FMPP-VQ are listed in Table 5.2 and Table 5.3. The operations in Table 5.2 are done according to the SIMD manner. These SIMD operations fall into two categories: numerical and logical operations and transfer operations. On addinner operation the other operand is supplied from the RW through the local data bus. On subtraction, an external data is given through the global data bus. Note that complement operation is performed as an exclusive-nor between the OW and zero value. The transfer operations moves data from one word to another. Before a numerical or logical operation is done, one of two operands should be transferred to the OW by one of *any*2operand operations. The search-flag oriented operations are related to the SF. On the search

min=All 1 # All bits are 1
$mask$=All 1 #(every bit is masked)
for $i = m + \lfloor \log_2 k \rfloor - 1$ **to** 0 # From MSB to LSB
  $mask[i] = 0$ # The target bit is unmasked.
  $min[i] = 0$
  search(min,$mask$)
  mrr(E_R,)# Confirmation
  if $E_R = 0$ **then**
    $min[i] = 1$
  endif
end
search(min,0) # Search the minimum value.
mrr(,$address$) # resolve the address of the PE

Figure 5.19: Program of the minimum value search.

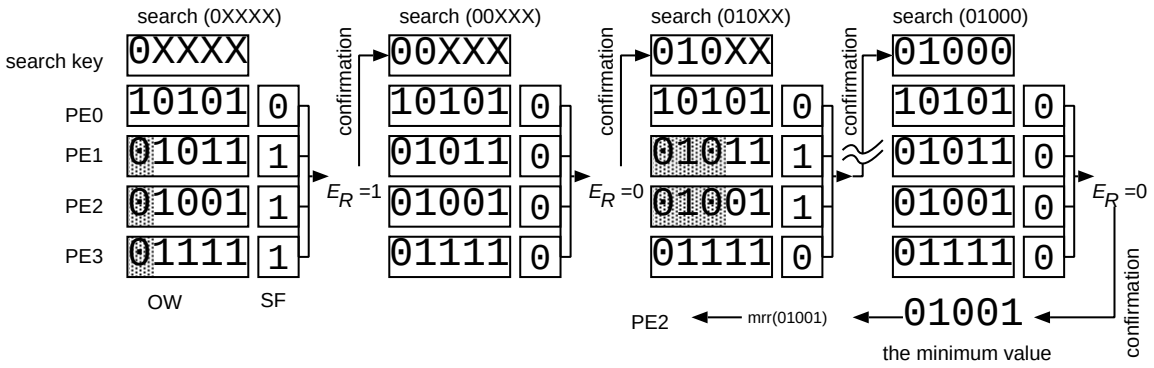


Figure 5.20: The minimum value search procedures.

operation a search key is given through the global data bus. It is simultaneously compared with all data in the operand words. An SF becomes true if it is equal to the search key. It is masked by the $mask$ signal. The mrr operation produces E_R which indicates whether there is any true search flag or not and also obtains the top most PE address whose SF is true. The writecode operation is used to write code vectors to the OW . Of course, the readcode operation to read out the OW is prepared, but it is not used on the nearest neighbor search. It is for test operations described in Section 5.5.4.

In the FMPP-VQ, each operation takes two clock cycles. Figure 5.21 shows the timing diagram of the FMPP-VQ. All the control signals to the PEs are activated at the first edge of the clock. It is deactivated at the second edge. The address or data inputs are fixed ahead of the first edge. It is held until the end of the second edges. It takes two clock cycles to perform an operation of the FMPP-VQ. In order to modify the values of a word by an internal word on addinner or transfer operations, the local data bus should be precharged prior to these operations, which takes two clock cycles. Thus

Table 5.2: All available SIMD operations of the FMPP-VQ.

operation	synopsis	#cycle
numerical and logical operations		
subtraction($data$)	OW $-$ $data$ $\rightarrow$ {OF, TW}	2
addinner	RW + OW + $\overline{\text{OF}}$ $\rightarrow$ TW	4
complement	$\overline{\text{OW}}$ $\rightarrow$ TW if OF =0	2
transfer operations		
code2operand(w)	transfer CW(w) to OW	4
temp2result	transfer TW to RW	4
result2operand	transfer RW to OW	4
temp2operand	transfer TW to OW	4

Table 5.3: Other operations of the FMPP-VQ.

operation	synopsis	#cycle
search-flag oriented operations		
search(key,$mask$)	SF $=1$ if OW $= key\&\overline{mask}$	2
mrr(E_R,$address$)	resolve the topmost PE whose SF is true. E_R becomes false if there is no true. SF $address$ is the address of the topmost PE.	2
To write a codebook to codebook words		
writecode($address$,$data$)	data write to CW(w)	2

these operation takes four clock cycles, while the operations where an operand is given from the global data bus such as subtraction does not need any precharge cycle.

Figure 5.22 shows the whole procedure to perform the nearest neighbor search. The initflag operation is required on the FMPP-VQ64 because of the design fault. The required number of clock cycles N_{NNS} is described in Equation (5.10).

$$N_{NNS} = t_{abs} + t_{minimumsearch} + t_{others} = 24k + 2(m + \lfloor \log_2 k \rfloor) + 8 \tag{5.10}$$

It takes 470 clock cycles when k=16 and m=12: 416 cycles to compute the absolute distance, 50 clock cycles to obtain the address where the nearest neighbor vector is stored and 4 clock cycles for the other operations. Note that the number of clock cycles does not depend on the number of PEs n (i.e., size of a codebook).

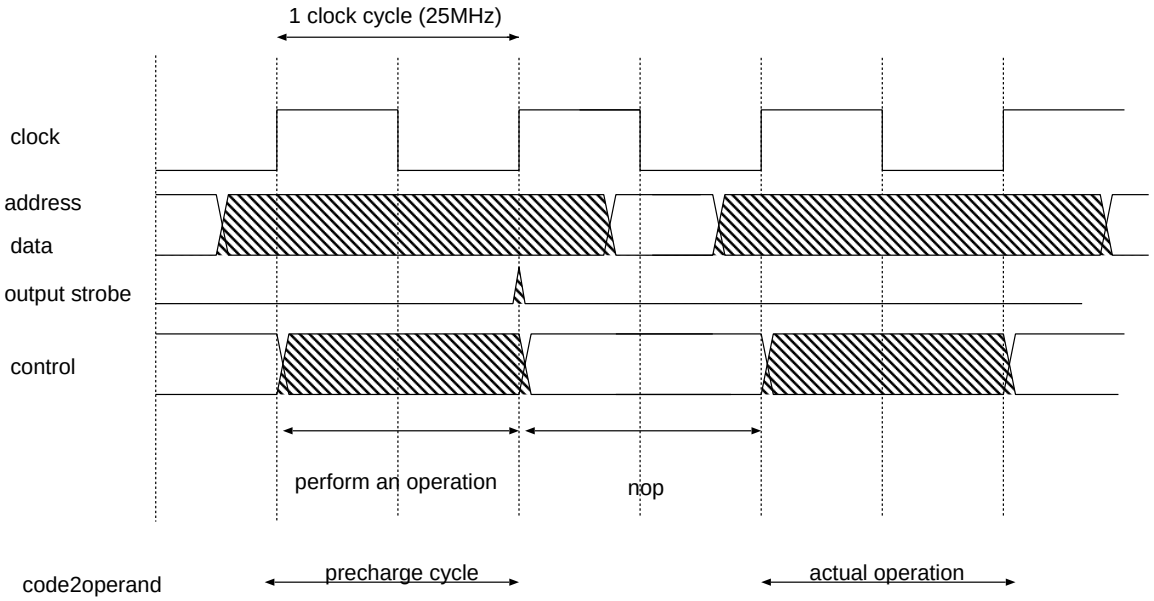


Figure 5.21: Timing Diagram of the FMPP-VQ.

for $i = 0$ to 15 # compute absolute distance
begin #416cycles=24*16
initflag()
code2operand(i)
subtraction(x_i)
temp2operand
} $|y_i - x_i|$
complement
temp2operand
addinner
} $\Sigma_{j=0}^{i-1}|y_j - x_j| + |y_i - x_i|$
temp2result end
result2operand # 4
$mask$=0b111111111111
$searchval$=0b000000000000
for $i = 11$ downto 0 # extract minimum value
begin # 48=12*4
$mask[i]$=0
search($searchval\ mask$)
mrr(E_R,)
$searchval[i]$=1 if E_R = 0
end
search($seachval$)# 2
mrr(,min_i) # 2

Figure 5.22: Whole procedure to perform the nearest neighbor search.

Table 5.4: LSI specifications of the FMPP-VQ4.

Process	0.7μm double-metal single-poly CMOS
Die size	26.3mm^2
Area for 4 PEs	1.12mm^2
# IOs	116
Power dissipation	3.8mW @(25MHz,5V)

5.5 Implementations of FMPP-VQ LSIs

In this section, two LSI implementations of the FMPP-VQ architecture are introduced. The first LSI contains four PEs called "FMPP-VQ4," which is for evaluating functionalities of the FMPP-VQ. Almost all functionalities are verified, but we found several faults. Secondly, we have designed and fabricated an LSI with 64 PEs called "FMPP-VQ64." It can be applied to low bit-rate image compression using vector quantization. It works properly at 25MHz and achieves high performance and low power.

5.5.1 An LSI Including Four PEs and TEGs: FMPP-VQ4

We have implemented an LSI called "FMPP-VQ4" including four PEs and some test circuitries using a 0.7 μm double-metal single-poly CMOS process. Four PEs and a 12bit sense amplifier are shown in the chip microphotograph of Figure 5.23. Table 5.4 shows specifications of the LSI. All the primitive control signals to the PEs and sense amplifiers are assigned to the primary input signals of the LSI, which increase the number of IO pins and chip area, while enhancing testability and controllability of the FMPP-VQ. All functionalities to perform the nearest neighbor search work properly at 25MHz, which is the same value obtained from circuit simulations. In the LSI, a 2bit ALU, a sense amplifier and 32bit carry chain are implemented. Unfortunately, they can not work correctly because of a short circuit between the power and ground nodes.

In the FMPP-VQ4, several faults are found. The most critical fault is as follows.

- At the code2operand operation, 4 MSBs of the operand word becomes unknown state. It is because the local bit lines of these bits are floated on the operation.

In the FMPP-VQ64, these 4 MSBs are connected to the ground node at the code2operand operation.

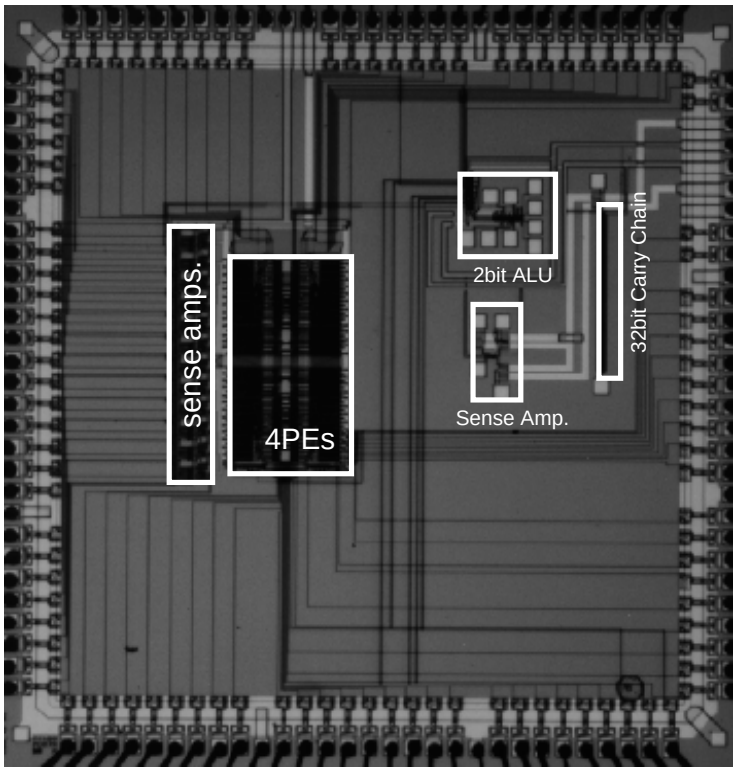


Figure 5.23: Chip microphotograph of the FMPP-VQ4.

5.5.2 An LSI Including 64 PEs and Control Logics: FMPP-VQ64

We have implemented and fabricated the FMPP-VQ64 LSI using the same 0.7μm CMOS process as that of the FMPP-VQ4. Figure 5.24 shows its detailed block digram. It is controlled by a 5-bit operation code supplied to the IO control logic ICL and the global control logic GCL. The control signals from the GCL to 64 PEs are doubled to decrease the load capacitance and to enhance the performance. Upper and lower 32 PEs are controlled by individual drivers. To equalize the distance from the ICL to every PE, the global data bus is laid out as the shape of an H character. To enhance its testability it has been designed such that all primitive control signals to the PE can be directly given from the IO pins. The number of IO pins is 50 except the primitive control and power supply pins. It is fully functional by the primitive control signals. But the control logics do not work correctly. In the FMPP-VQ4, all the circuitries can be simulated by the transistor level simulator HSPICE^{TM}, while in the FMPP-VQ64 the transistor level simulation is impossible since the number transistors amounts to several hundred thousand. To simulate the whole circuitries of the FMPP-VQ64, we have developed a behavioral HDL description of the PE. Figure 5.25 shows the Verilog-HDL description of the operand word and the carry chain. The whole circuit of the FMPP-VQ64 is simulated in the logic level by the logic simulator Verilog-XL. The fault of the control logics occurs because the HDL description of the PE does not follow their accurate behavior.

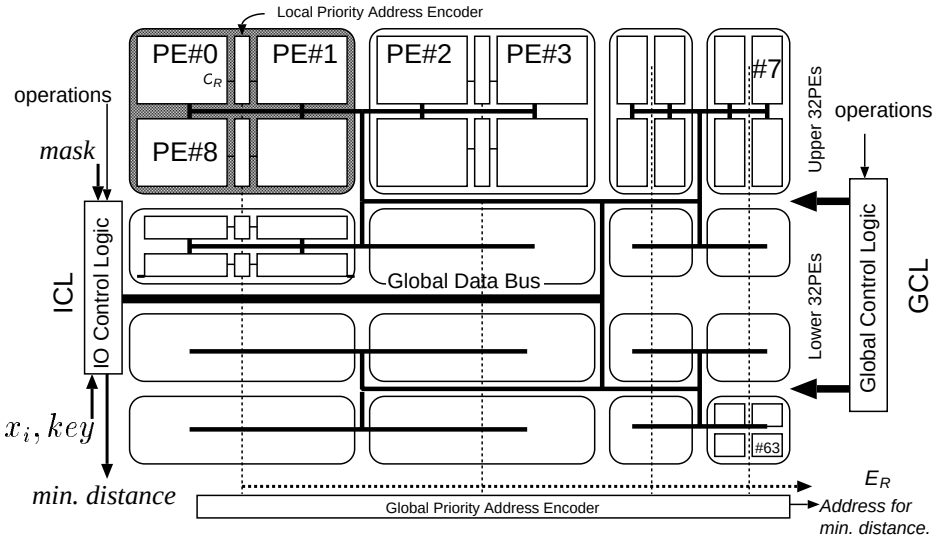


Figure 5.24: The detailed block diagram of the FMPP-VQ64.

Figure 5.26 shows the chip microphotograph. LSI Specifications of both the FMPP-VQ4 and FMPP-VQ64 are described in Table 5.5. Figure 5.27 shows a measured Shmoo plot of supply voltage versus cycle time. While the recommended supply voltage of the target process is 5.0V, the FMPP-VQ64 LSI works properly at 3.0V/25MHz and at 2.5V/20MHz. The power dissipation of the FMPP-VQ64 is 20mW under the condition of 3.0V/25MHz.

Figure 5.28 shows the dynamic current flow of the power-supply pin during an iteration of the absolute distance computation. The operating condition is at 25MHz/5V. The current flow has peaks when operations such as subtraction, complement and addinner are executed.

Table 5.5: LSI specifications of both of the FMPP-VQ4 and FMPP-VQ64.

	FMPP-VQ4	FMPP-VQ64
Process	0.7μm double-metal single-poly CMOS	
Die size	26.3mm^2	52.6mm^2
# IOs	116	148
Power dissipation	3.8mW@(25MHz,5V)	9.30mW@(20MHz,2.5V) 20.5mW@(25MHz,3.0V) 128mW@(25MHz,5.0V)
Area of a PE array	2.43mm^2(2×2)	23.5mm^2(8×8)
Area of a PE	.28mm^2	.37mm^2

```
module operandword (c1_, sum, pp, b0, b1, search, sin, c0_, w0, w1, we);
    output c1_,sum,pp;
    inout b0,b1;
    inout search,sin,c0_,w0,w1,we;
reg q,c1;
    always @(b0,w0,we)
#10
                case(b0,w0,we)
                        3'b011: q=0;
                        3'b111: q=1;
                endcase
        not     not0(we_,we);
        and and0(write0,w0,we_);
        and and1(write1,w1,we_);
        bufif1 (weak0,weak1) buf0(b0,q,write0);
        notif1 (weak0,weak1) not1(b1,q,write1);
        not not2(c0,c0_);
        xor xor0(sum,q,b0,c0);
or  or0(pp,x0,x1);
not not4(q_,q);
not not5(b1_,b1);
not not6(b0_,b0);
and and2(x0,q,b0_,b1);
and and3(x1,q_,b0,b1_);
        // adder
        always @(b0,c0,q)
                begin
                        c1=(b0&&c0)||(b0&&q)||(q&&c0);
                end
        not not3(c1_,c1);
        // search
bufif1 bufsearch0(search,sin,pp);
endmodule
```

Figure 5.25: Verilog-HDL description of the operand word and the carry chain.

As described in Section 5.4.5, the FMPP-VQ computes the nearest neighbor search (NNS) in 470 clock cycles. Thus, the FMPP-VQ64 completes the NNS in 18.8μsec at 25MHz. It can perform 53,000 NNSs per second.

In order to eliminate unnecessary state changes in the ALU, the inverter in Figure 5.10 is only activated at the numerical operations. To evaluate the effect, power dissipations are measured. Table 5.6 shows the results. Unfortunately, activating the inverter only at the numerical operation increases the power. It may be because the input nodes of the inverter has high probability of high voltage. When the inverter is deactivated and the input node of inverter is high, the output node is floated, which leads short-circuit current in the XNOR gate or the carry chain. To eliminate the short-circuit current, the inverter must be deactivated by both of NMOS and PMOS FETs. But it makes the circuit area larger. Therefore, in the FMPP-VQ64M, which is the modified version of the FMPP-VQ64, the inverter is controlled by a PMOS FET (See Section 5.6.3). Since the input node of the inverter has

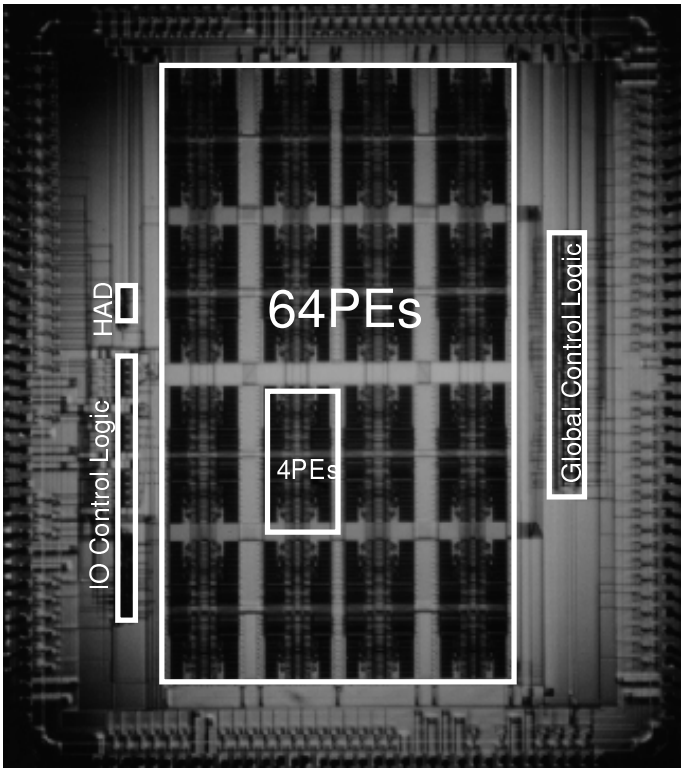


Figure 5.26: The chip microphotograph of the FMPP-VQ64.

high probability of high voltage, the output node of the inverter has high probability of discharged state causing no short-circuit current in the subsequent circuits when deactivating the inverter.

Table 5.6: Comparisons of power dissipation by activating the inverter at the numerical operation and by always activating the inverter. The condition is 5V/25MHz.

Activation	at the numerical operations	always
Power dissipation	193mW	128mW

5.5.3 Integration Density of the FMPP-VQ64

Here, we evaluate the integration density of the FMPP-VQ64 compared with a conventional SRAM designed with the same technology. Table 5.7 shows the area of a single PE of the FMPP-VQ64 with the area of the 8kbit SRAM designed with the same 0.7μm process[DAT96]. The 64PEs of the FMPP-VQ64 contains 8kbit codebook words, which area is 9.5 times larger than the conventional SRAM. Looking at Figure 5.5, the global data bus, control lines and address bus occupies about a half of PEs. Without these bus and control lines, the area of the 64PEs is reduced to 10.1mm^2, which is only 4 times larger than the conventional SRAM. The reason why the area becomes twice including

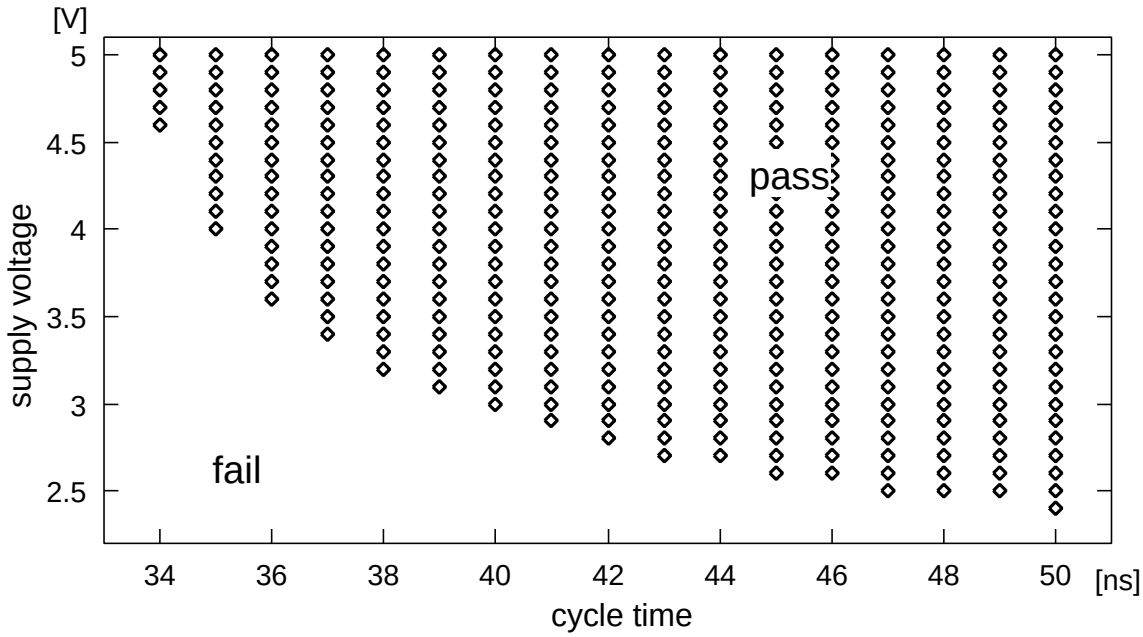


Figure 5.27: A measured Shmoo plot of supply voltage versus cycle time in the FMPP-VQ64.

these bus and control lines is mainly that the fabricated process has only two metal layers. If the brand-new sub-micron multiple-metal-layer process is available for use, the integration density of the FMPP will increase considerably.

Table 5.7: Area for 1 PE of the FMPP-VQ64 and 8kbit SRAM fabricated by the same 0.7μm process.

	area(mm^2)	
64PEs		23.5
1PE	total	.367
	16codebook words+ALU	.137
	priority address encoder	.02
8kbit SRAM		2.45

5.5.4 Testability of the FMPP-VQ64

To enhance the testability of logic LSIs scan methods have commonly been used. The FMPP-VQ64 adopts the parallel random-access scan methodology[WE93] owing to its memory-based structure. Codebook words can be accessed in the same manner as conventional RAMs. Furthermore, all words and flags in the ALU can be accessed with read/write operations.

Figure 5.29 illustrates the address control scheme of the FMPP-VQ64. It has 10-bit address

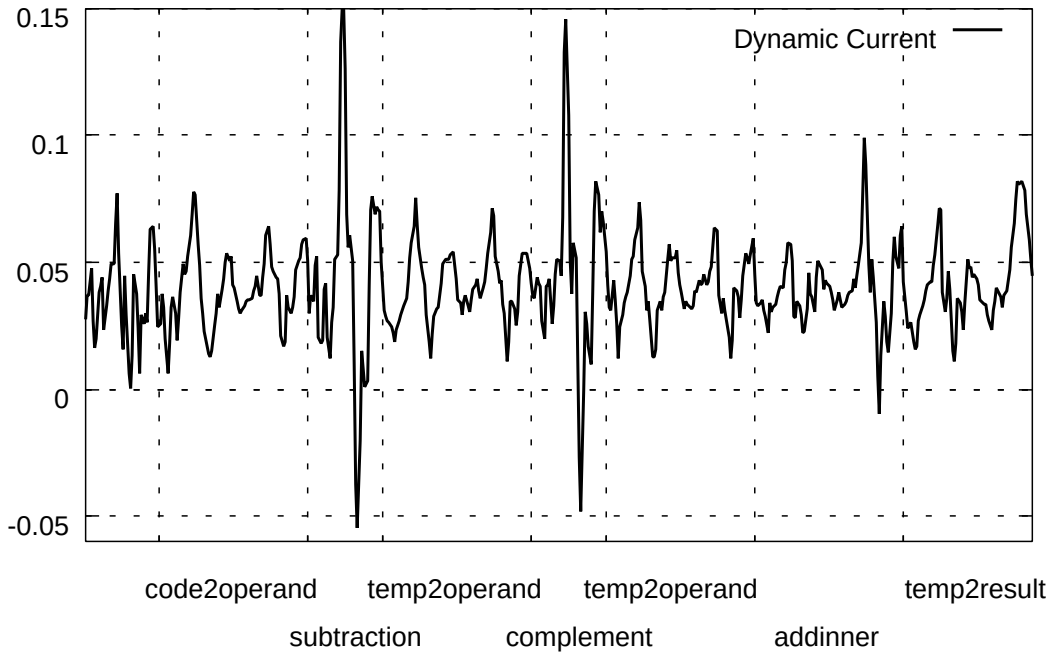


Figure 5.28: Dynamic Current flow on the absolute distance computation.

control lines for 1k (16×64) codebook words. Higher 6-bit signals to the higher address decoder (HAD) represent a PE address. Lower 4-bit signals to the lower address decoder (LAD) are used for a codebook-word address in each PE. On SIMD operations, such as subtraction, the input signal $rcall$ is set to high. Then all higher decoded addresses become active. All PEs work simultaneously according to the SIMD control method. On read/write operations, the higher decoded address determines a single PE. In the local control logic (LCL) shown in the bottom left of Figure 5.29, a higher decoded address from the HAD enables the local decoded address (LDA) from the LAD. To append parallel random-access capability to the ALU, the higher decoded address also enables control signals from the global control logic.All words and flags in the ALU can be accessed through read/write operations, which improves the testability of the FMPP-VQ64 considerably. The number of signals controlled by a higher decoded address is only 10 in the ALU. The hardware overhead is very small. Note that the flags are connected to the local data bus as shown in Figure 5.4 in order to be read or written directly through the bus.

5.6 Modified Version of the FMPP-VQ: FMPP-VQ64M

The FMPP-VQ64 achieves both of high performance and low power. But it has some drawbacks to compute the NNS as follows. Eight operations are required to compute the absolute distance of a single dimension, as already shown in Figure 5.22. Half of these 8 operations transfer a value from one word to another such as temp2operand. These transfer operations decrease the throughput

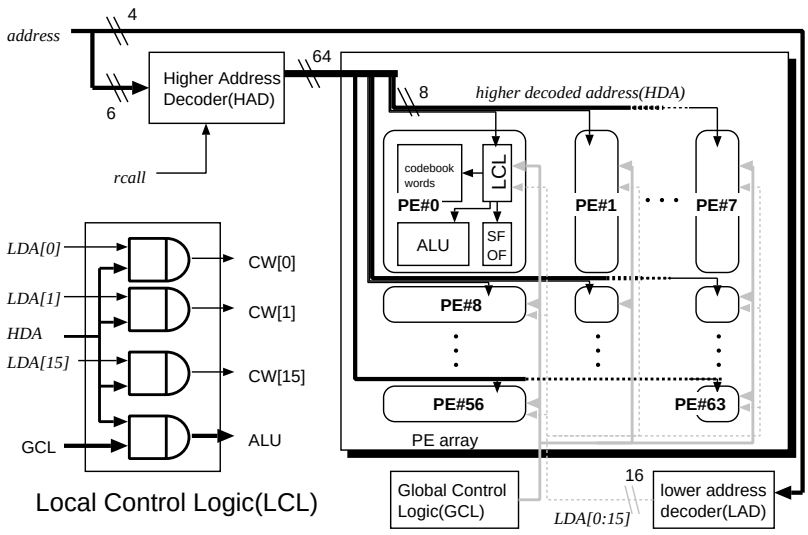


Figure 5.29: Parallel random-access capability to the ALU.

and increase the power consumption. The number of operations can be reduced if operation results are directly written to the OW . The optimal transistor-sizing may reduce the short-circuit power dissipation. In this section, a modified-version of the FMPP-VQ64 is described, which is called "FMPP-VQ64M."

The structure of the FMPP-VQ64M is similar to the FMPP-VQ64. But it can perform 91,000 NNSs per second which is almost two times faster than the FMPP-VQ64. It integrates a highly-functional control logic to manage the nearest neighbor search.

5.6.1 Structure of a PE

Figure 5.30 compares two PE structures of FMPP-VQ64 and FMPP-VQ64M. Codebook words are 16 words of eight-bit conventional SRAM cells, which store a 16-dimensional code vector. The ALU computes the absolute distance. The operand word receives data from the local data bus and performs logical operations. In the FMPP-VQ64, operation results from the carry chain or the operand word are temporarily written to the temporary word. To reuse the value stored in the temporary word, it must be transferred to the operand word, which always consumes an operation. To remove such drawbacks, the ALU of the FMPP-VQ64M does not have the temporary word. Operation results can directly be written to the operand word. It can compute the absolute distance for an element x_i of $\vec{x}$ in four steps, while the FMPP-VQ64 takes eight steps.

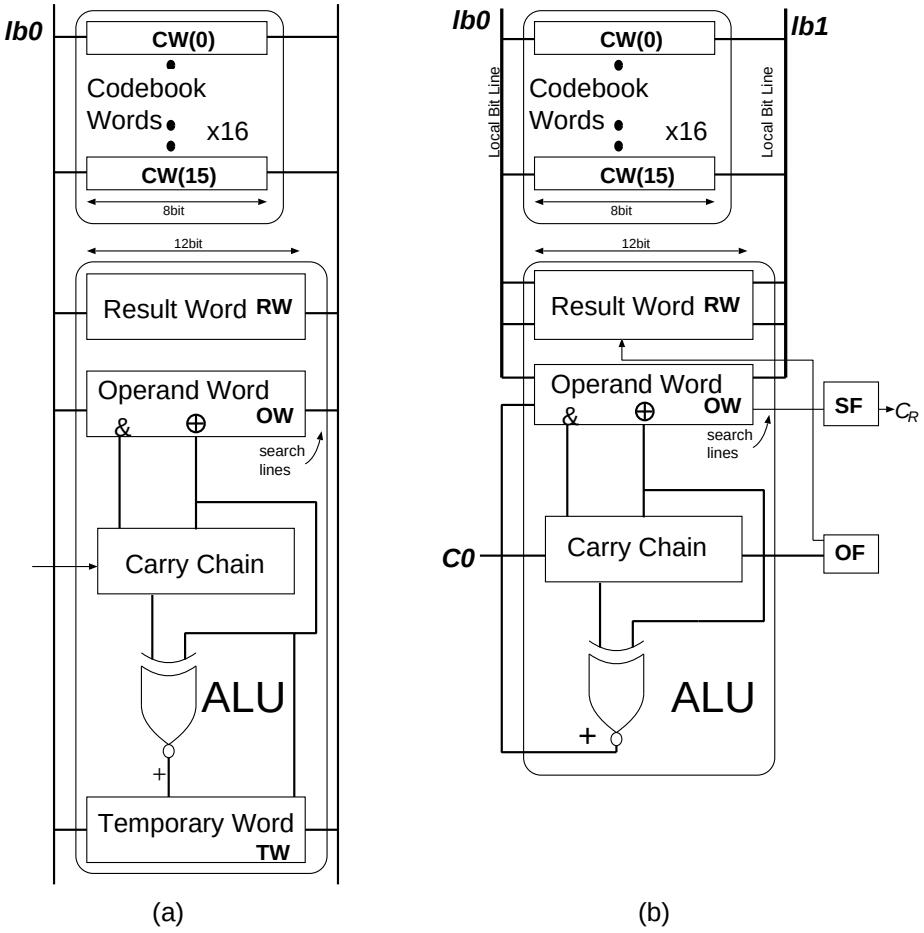


Figure 5.30: PE structures of the FMPP-VQ64(a) and the FMPP-VQ64M(b).

5.6.2 Absolute Distance Computation

In the FMPP-VQ64M, the absolute distance is computed element by element according to Equation (5.11).

$$\sum_{l=0}^{j} |x_l - y_{il}| = \begin{cases} \sum_{l=0}^{j-1} |x_l - y_{il}| + y_{ij} - x_j & (y_{ij} \geq x_j) \\ \overline{\{(\sum_{l=0}^{j-1} |x_l - y_{il}|) + y_{ij} - x_j\}} & (y_{ij} < x_j) \end{cases} \tag{5.11}$$

It requires a conditional operation according to the results of $y_i - x_i$, which is enabled by the overflow flag OF. Three operations as shown below compute the absolute distance.

Operation 1 Compute $y_{ij} - x_j$

Operation 2
- Condition 1 $[(y_{ij} - x_j \geq 0)]$ Accumulate $y_{ij} - x_j$ to $\sum_{l=0}^{j-1} |x_l - y_{il}|$.
- Condition 2 $[(y_{ij} - x_j < 0)]$ Accumulate $y_{ij} - x_j$ to $\overline{\sum_{l=0}^{j-1} |x_l - y_{il}|}$.

Operation 3	Condition 1	$[(y_{ij} - x_j \geq 0)]$	Transfer the original value to the RW .
	Condition 2	$[(y_{ij} - x_j < 0)]$	Transfer the inversed value to the RW .

These operations are performed in 4 steps as below.

Step1: [code2operand(j)] The jth element of $\vec{y}_i$, y_{ij} is sent to the OW.

Step2: [subtraction(x_j)] The jth element of $\vec{x}$, x_j is broadcast to all PEs. Then, the subtraction, $y_{ij} - x_i$ is done. The result is stored in the OW. The overflow value is stored in the OF.

Step3: [addinner]

$$\begin{array}{ll} (y_{ij} - x_j) \geq 0 & \overline{\sum_{l=0}^{j-1} |x_l - y_{il}| + \mathsf{OW}} \rightarrow \mathsf{OW} \\ (y_{ij} - x_j) < 0 & \sum_{l=0}^{j-1} |x_l - y_{il}| + \mathsf{OW} \rightarrow \mathsf{OW} \end{array}$$

Step4: [temp2result]

$$\begin{array}{ll} (y_{ij} - x_j) \geq 0 & \mathsf{OW} \rightarrow \mathsf{RW}. \\ (y_{ij} - x_j) < 0 & \overline{\mathsf{OW}} \rightarrow \mathsf{RW}. \end{array}$$

At **Step1**, an element of code vectors is tranfer to the operand word. There exists no useless transfer operation between the other operations. At **Step2**, $\overline{x_j}$ is given through the local data bus and subtraction $y_{ij} - x_j$ is done. The overflow value is stored to the OF . If $y_{ij} - x_j \geq 0$, the OF becomes 1. At **Step 3** and **4**, the summation $\sum_{l=0}^{j-1} |x_l - y_{il}|$ is accumulated to $|x_j - y_{ij}|$. The value of the OF controls the operations at these two steps. The final value stored in the RW becomes $\sum_{l=0}^{j} |x_l - y_{il}|$ at the both conditions.

Figure 5.31 depicts the data flow to compute Equation (5.11) (in page 80). It shows two conditions according to the results of the subtraction in **Step2**. The upper condition is where $y_{ij} - x_j \geq 0$ and the lower one is the opposite state. In **Step1**, code2operand transfers y_{ij} from CW(j) to OW . In Step2 all PEs receive an element x_j of an input vector $\vec{x}$. In **Step 2'**, the subtraction between x_j and the OW is performed and the results are written to the OW and the OF . In **Step 3** addition between the RW and the OW is done and the result is written to the OW . The value from the RW to the local data bus is changed according to the OF . In **Step 4**, the value in the OW is moved to the RW , which becomes $\sum_{l=0}^{j} |x_j - y_{ij}|$.

5.6.3 Detailed Structure of the ALU

Figure 5.32 shows a detailed schematic diagram of two-bit slice of the ALU. Figure 5.33 shows a detailed schematic diagram of the operand word. When an operation between the OW and an operand from the local bit lines is done, the result is written to the OW itself. To write back the operation

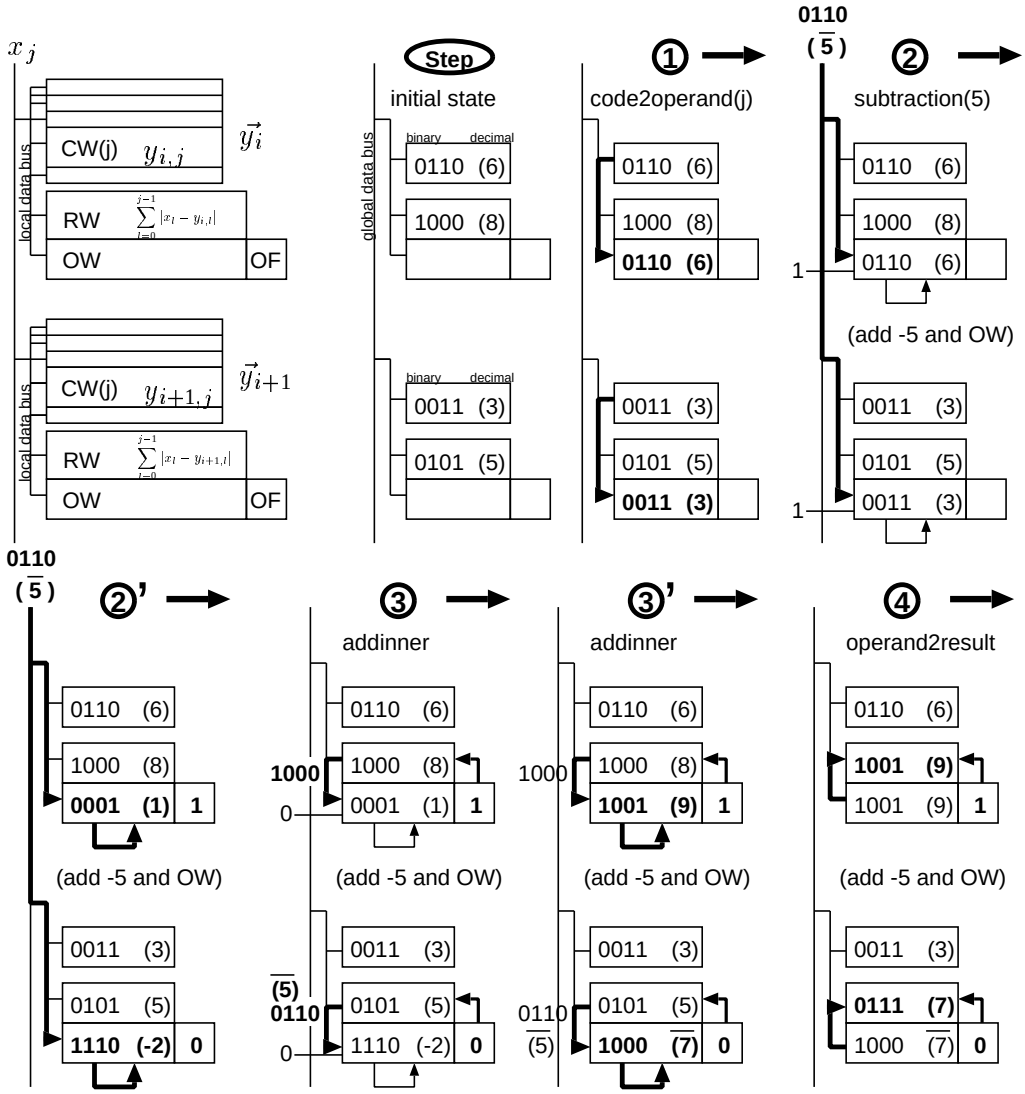


Figure 5.31: A single dimension slice of the absolute distance computation in FMPP-VQ64M.

result to the OW safely, two CMOS switches Sw0 and Sw1 are cut off. The gate capacitance of nodes T and $\overline{T}$ holds the value stored in the OW .

In the result word RW (See Figure 5.35), W0 and W1 are controlled by the value of the OF, which enables read and write operations of the inversed value required at **Step3, 4**. The controlled inverter in Figure 5.34 decreases power consumption considerbly. The control signal $\overline{\text{control}}$ is activated on the numerical operation. As already written in Section 5.5.2, the input node of the inverter has a high probability of high voltage. Once its output node is discharged, it can not be re-charged unless $\overline{\text{control}}$ is activated. THus, the XNOR gate and the carry chain dissipate power at numerical operations.

Table 5.8 compares the specifications of the PEs of FMPP-VQ64 and FMPP-VQ64M. The number of transistors in the FMPP-VQ64M is decreased to 86% compared to that in the FMPP-VQ64, while the area of PE is almost same. It it mainly because the number of vertical signals in the ALU increases.

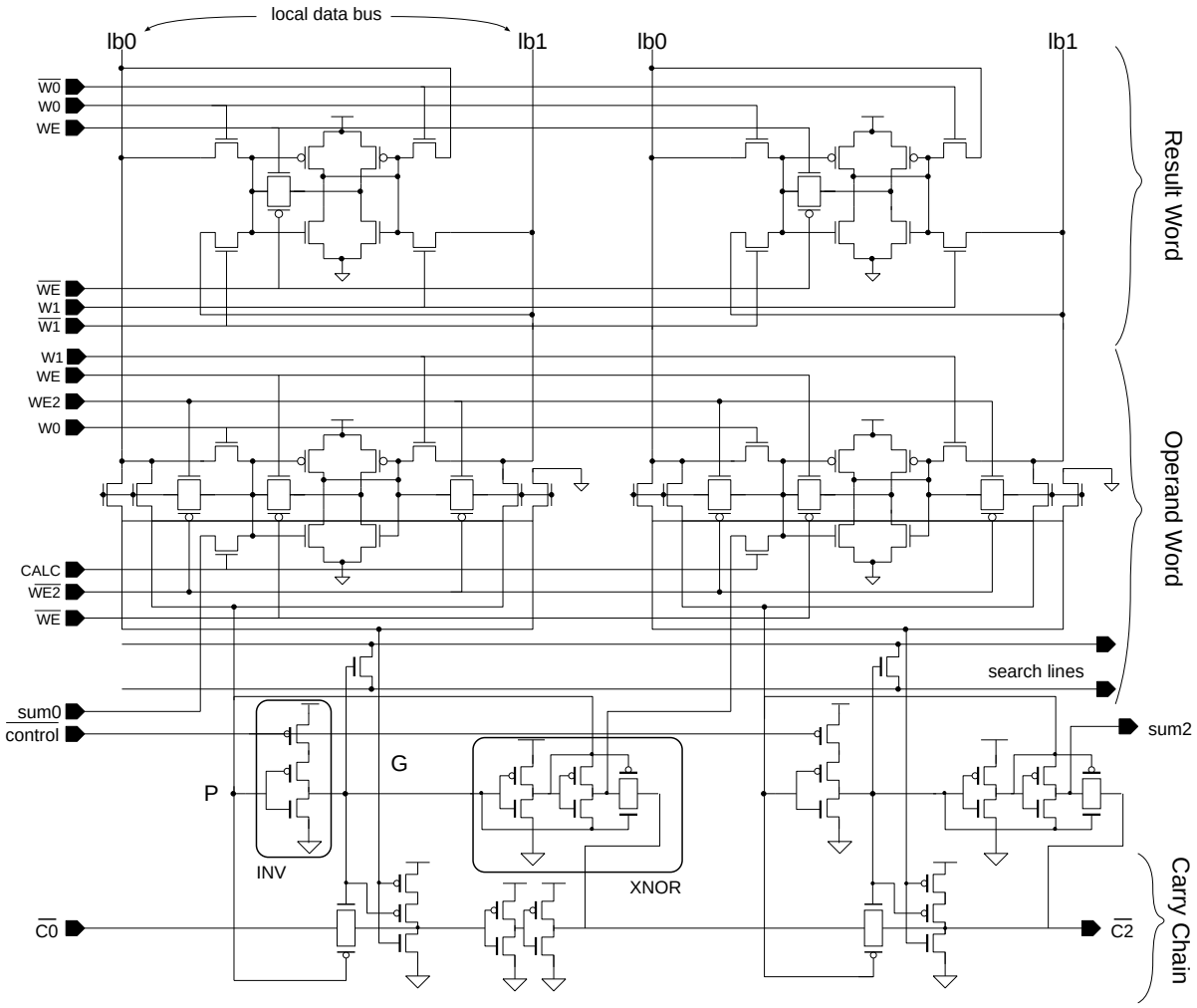


Figure 5.32: Two-bit slice of the ALU.

The total number of transistors in the PE is increased, since the local control logic to manage W0 and W1 in the RW becomes complecated. The number of clock cycles to compute the absolute distance, however, is decreased to 53% of the FMPP-VQ64.

5.6.4 A Highly-Functional Control Logic

The FMPP-VQ64 has a primitive control logic which translates a specified 5-bit control code to primary input signals of the PE array. In the FMPP-VQ64**M**, a highly-functional control logic is implemented. It receives a start signal and automatically performs the nearest neighbor search. It is called the auto-execution mode. To enhance its testability, all primary input signals are directly controlled in the primary control mode. In order to observe some temporary values which are on the way of operations, a HALT signal is given to the control logic. There exist two modes in the auto exectution mode. One is the normal operation mode and the other is the fast operation mode. In the normal operation mode, all operations are done in a single cycle. On the other hand, in the fast

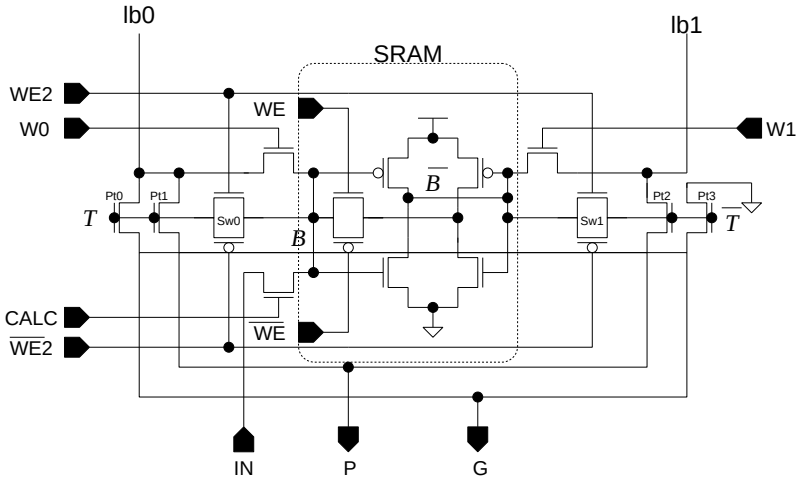


Figure 5.33: Structure of the operand word.

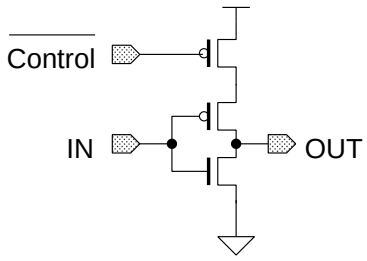


Figure 5.34: Inverter controlled by a PMOS FET.

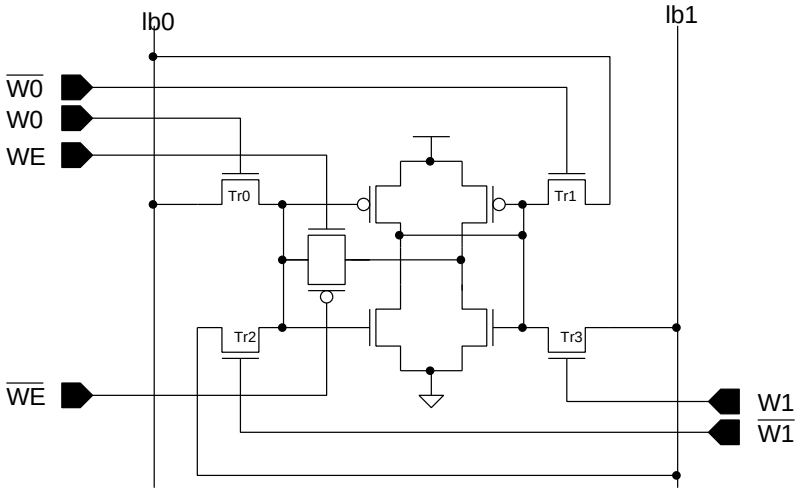


Figure 5.35: Structure of the result word.

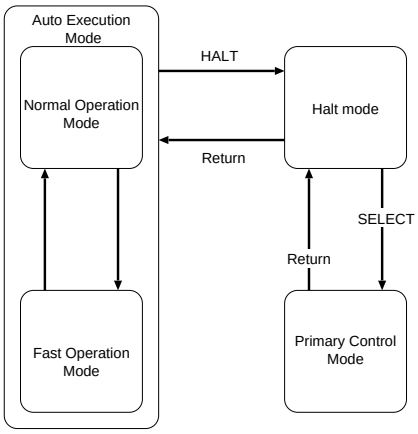


Figure 5.36: The flow of mode changes in the FMPP-VQ64M.

opearation mode, numerical operations are done in two cycles. In Figure 5.31, numerical operations such as subtraction and addinner can be divided two phases. The second phase is denoted by a single quotation mark. At the first phase, an operand is given to the operand word. At the second phase, the result is written to the operand word. The numerical operations take twice as long as the other non-numerical operations. If the numerical operations are done in two cycles, the clock cycle can be shortened. Thus, the whole operation can be done faster. The detailed simulation results and performance estimation are given afterwards.

5.6.5 Specification and Implementation

Equation (5.12) and (5.13) show the number of clock cycles to compute the NNS in the normal operation mode and the fast operation mode respectively.

$$N_{normal} = 14 \times 16 + 50 = 274 \quad (5.12)$$

Table 5.8: Comparison of areas and performance for the FMPP-VQ64 and the FMPP-VQ64M.

	FMPP-VQ64	FMPP-VQ64M
# of Tr. per bit in the ALU	45	39
Area of a PE	.241mm^2	.238mm^2
# Trs of a PE	1524	1534
# of clock cycles to compute the absolute distance of a single dimension	26	14

$$N_{fast} = 18 \times 16 + 50 = 338 \tag{5.13}$$

We estimate that the FMPP-VQ64M works at 25MHz (40ns.) at the normal operation mode. The numerical operations on the FMPP are 1.5 times slower than the other operations. The operation speed can be 1.5 times faster in the fast operation mode, resulting 37.5MHz clock frequency. Thus, the FMPP-VQ64M performs 91,000 NNSs per second at the normal operation mode, while it performs 111,000 NNSs per second at the fast operation mode.

Table 5.9 lists power consumption values of a PE from circuit simulations, which shows the effect of two optimizations compared with the FMPP-VQ64. First, the transistor size is optimized to reduce the short-circuit current, which decreases the power by 6%. The controlled inverter decreases the power by 23%. Table 5.10 shows the dissipated power in each part of the PE. It proves that the carry chain dissipates half of the total power. The FMPP-VQ64 activates the carry chain three times in a single dimensional absolute distance computation, while the FMPP-VQ64M activates it twice.

Table 5.9: Power consumption of the FMPP-VQ64M from circuit simulations of a PE at 25MHz 5.0V.

	Optimization Method	Power
FMPP-VQ64		3.03mW
FMPP-VQ64M	before optimizing Tr. size	2.01mW
	after optimization	1.86mW
	activate INV at operations	1.45mW

The power consumption expected from that of the FMPP-VQ64 is shown in Table 5.11. The power consumption of the FMPP-VQ64M decreases by half, while the number of operations for the nearest neighbor search becomes twice of the FMPP-VQ64's. Thus, its total energy consumption becomes 1/4 compared with the FMPP-VQ64.

Table 5.10: Power dissipation map for all the components in a PE.

whole PE	1.45mW	100%
Carry Chain	0.82mW	56%
Operand Word	0.32mW	22%
XNOR	0.07mW	5%
Result Word	0.06mW	4%
Others	0.11mW	11%

Table 5.11: The power consumption of the FMPP-VQ64M expected from the measured results of the FMPP-VQ64.

Clock Freq.	28.5MHz	25MHz	20MHz
Supply Volt.	5.0V	3.0V	2.5V
Power(mW)	65	9.9	4.5

The FMPP-VQ64M is designed using the same double-metal single-poly 0.7μm CMOS process as that of the FMPP-VQ64. The layout pattern of the PE is shown in Figure 5.37. At the first stage when we start to design the FMPP-VQ64M, we expected that the area for a PE would be smaller than that of the FMPP-VQ64, since the PE of the FMPP-VQ64M has no temporary word. But the area of the PE becomes almost the same. Table 5.12 summarizes the number of transistors and area for each part in the PEs. The areas for the operand word and the result word in the FMPP-VQ64M becomes larger than those in the FMPP-VQ64.

Table 5.12: Areas for PEs of the FMPP-VQ64 and FMPP-VQ64M.

	FMPP-VQ64M		FMPP-VQ64	
	# of Tr/bit.	Area/bit	# of Tr/bit.	Area/bit
Codebook Word	6	433.4μm^2	6	433.4μm^2
Result Word	10	840.0μm^2	8	710.1μm^2
Operand Word	17	1714μm^2	15	977.4μm^2
Carry Chain	11	695.5μm^2	11	695.5μm^2
XNOR	6	571.2μm^2	6	571.2μm^2
Temporary Word	N/A	N/A	10	930.7μm^2
Others	2	168.0μm^2	2	168.0μm^2
One bit slice of a PE (for 8LSBs)	142	11205μm^2	148	11203μm^2
Total (12bit) of a PE	1534	0.214mm^2	1522	0.208mm^2

Figure 5.38 shows the chip micrograph. The die size of the FMPP-VQ64M is 52.7mm^2. Table 5.13 compares the areas of FMPP-VQ64 and FMPP-VQ64M. FMPP-VQ64M actually integrates more complex control logics than that of the FMPP-VQ64. Table 5.14 summarizes the number of implemented standard cells in each control logic. These areas are almost same with both implemen-

tations. It is because the strategies of place and route are changed. The FMPP-VQ64 is placed and routed from the bottom to the top level. Each component such as the PE array or control logics are placed and routed respectively. Then the top level layout is created using these macro blocks. In the FMPP-VQ64M, however, full-custom components such as the PE array or sense amplifiers are placed as macro blocks, but the other random-logic cells in the control logics are placed and routed in the top level with these macro blocks. This strategy decreases the area of the control logic.

The FMPP-VQ64M is now under test. Test results will be shown in later.

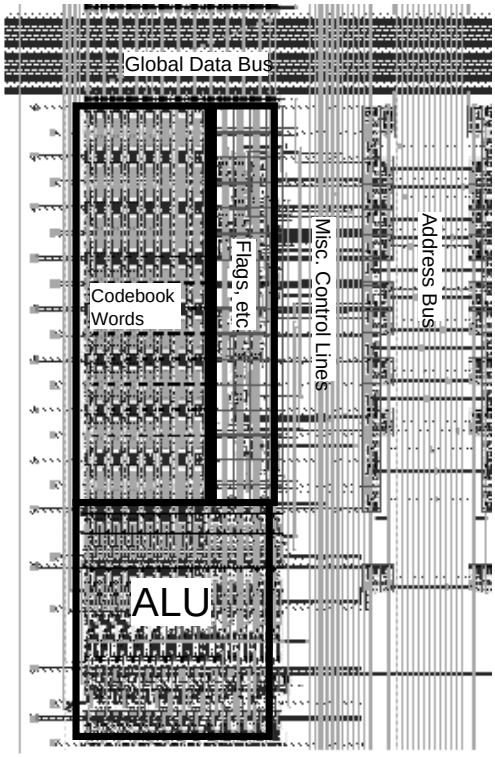


Figure 5.37: Layout of a PE.

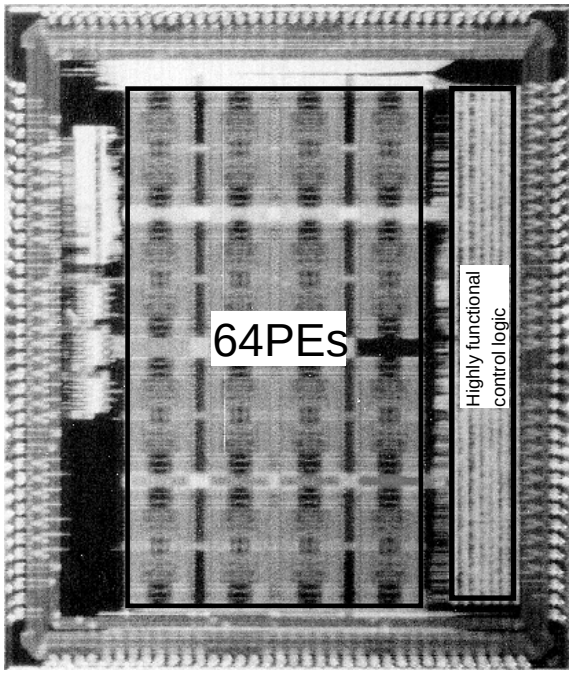


Figure 5.38: Chip micrograph of the FMPP-VQ64M.

Table 5.13: The areas for FMPP-VQ64 and FMPP-VQ64M.

	FMPP-VQ64M		FMPP-VQ64	
Die size (Including I/O PADS)	$6.63 \times 7.94\text{mm}^2$	52.64mm^2	$6.60 \times 7.86\text{mm}^2$	51.84mm^2
Core size (Without I/O PADS)	$5.78 \times 7.09\text{mm}^2$	40.98mm^2	$5.75 \times 7.01\text{mm}^2$	40.30mm^2
64 PEs	$3.70 \times 6.35\text{mm}^2$	23.50mm^2	$3.52 \times 6.24\text{mm}^2$	21.97mm^2

Table 5.14: Number of standard cells for control logics.

Logics	FMPP-VQ64M	FMPP-VQ64
D-FF	313	45
Other sequential logic cells	1956	548
Total	2269	593

5.7 Comparison with Other Implementations

In this section, several comparisons between the FMPP-VQ with the other implementations are given.

5.7.1 Comparison with the Other Vector Quantizer.

Here, the FMPP-VQ is compared with the other vector quantizers. Parameters for vector quantization are given again:

- k dimension of vectors
- N number of code vectors
- m bit width of vectors

The compared points are as follows.

Codebook optimization It is to hard to obtain a generalized codebook which can be applied to any type of images. It should be optimized for every frame of image. Thus, a codebook has to be updated in real-time.

Accuracy of the nearest vector If the vector from the NNS is a suboptimal one, the distance becomes larger, which decreases the quality of a reconstructed image.

IO bandwidth A large IO bandwidth enlarges the area and causes problems to mount the LSI on a circuit board.

Power Consumption Power consumption should be minimized for mobile telecommunication.

There are two algorithms to search the nearest neighbor vector. The tree-searched VQ (TSVQ) has less codebook search complexity in proportion to the logarithmic order of the size of a codebook than the full-searched VQ (FSVQ) adopted in the FMPP-VQ. But the size of memory to store a codebook becomes large. As for codebook optimization, the TSVQ has a major drawback that the

codebook cannot be optimized in real time, since it requires huge computation complexity to generate the tree-structured codebook.

Fang et al. proposed a systolic binary tree-searched vector quantizer[FCS+94]. Its block diagram is drawn in Figure 5.39. In conventional TSVQ implementations, the total size of a codebook reaches O($\sum \prod_v N_v \cdot v$). The value v means the number of levels and N_v is the number of code vectors in each sub-codebook. They reduce the size of a sub-codebook at each level into two, which results $2(n-1)$ code vectors in all the sub-codebook. The index of the nearest code vector in a sub-codebook is computed using the MSE as follows.

$$\begin{aligned} D_{v1} - D_{v0} &= |\vec{x} - \vec{y_{v1}}|^2 - |\vec{x} - \vec{y_{v0}}|^2 \\ &= \vec{y_{v1}}^2 - \vec{y_{v0}}^2 - (\vec{y_{v1}} - \vec{y_{v0}})\vec{x} \\ \text{if} \quad & D_{v1} - D_{v0} > 0 \\ \text{then} \quad & index[v] = 0 \\ \text{else} \quad & index[v] = 1 \end{aligned}$$

The terms of $\Delta = \vec{y_{v1}}^2 - \vec{y_{v0}}^2$ and $\delta = \vec{y_{v1}} - \vec{y_{v0}}$ are formerly prepared to guarantee real-time encoding with the small hardware.

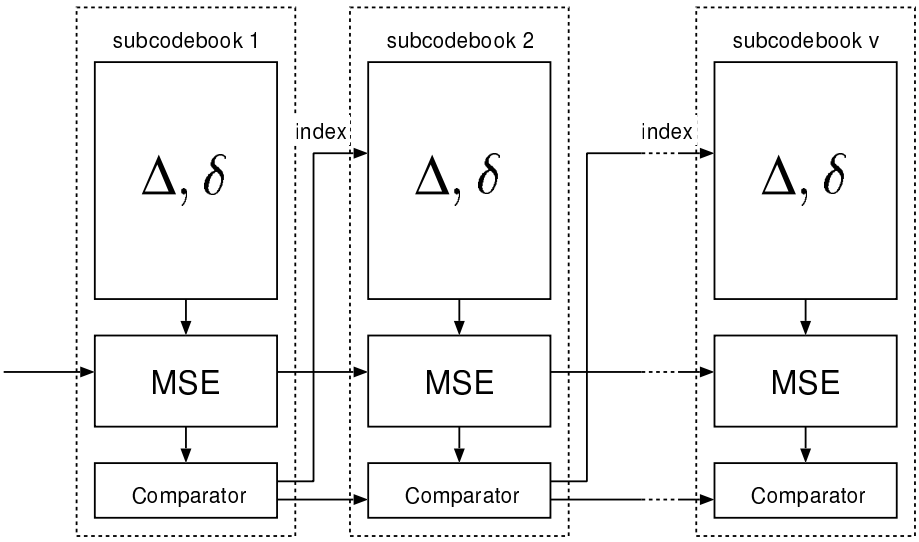


Figure 5.39: A systolic binary-searched vector quantizer.

Wang et al. propose a systolic array processor as shown in Figure 5.40[WC95]. It has N PEs that compute MAEs and N/k PEs that compare the MAEs. The latency is O($k+N$). The minimum distance can be obtained every k clocks. But It has some drawbacks. First, it has a large bandwidth of O(km), which amounts to 128 when $k = 16$ and m=8. Secondly, it contains a large number of latches inproportion to O(N) to store temporary values among PEs. In addition to that, they assume

100MHz clock frequency for both the inner and outer LSI. It is difficult to give data at such high clock frequency from outside of the LSI. As discussed in Section 5.7.3, the SRAM module implemented inside the LSI using the 0.7μm process for the FMPP-VQ LSIs cannot provide data at 100MHz.

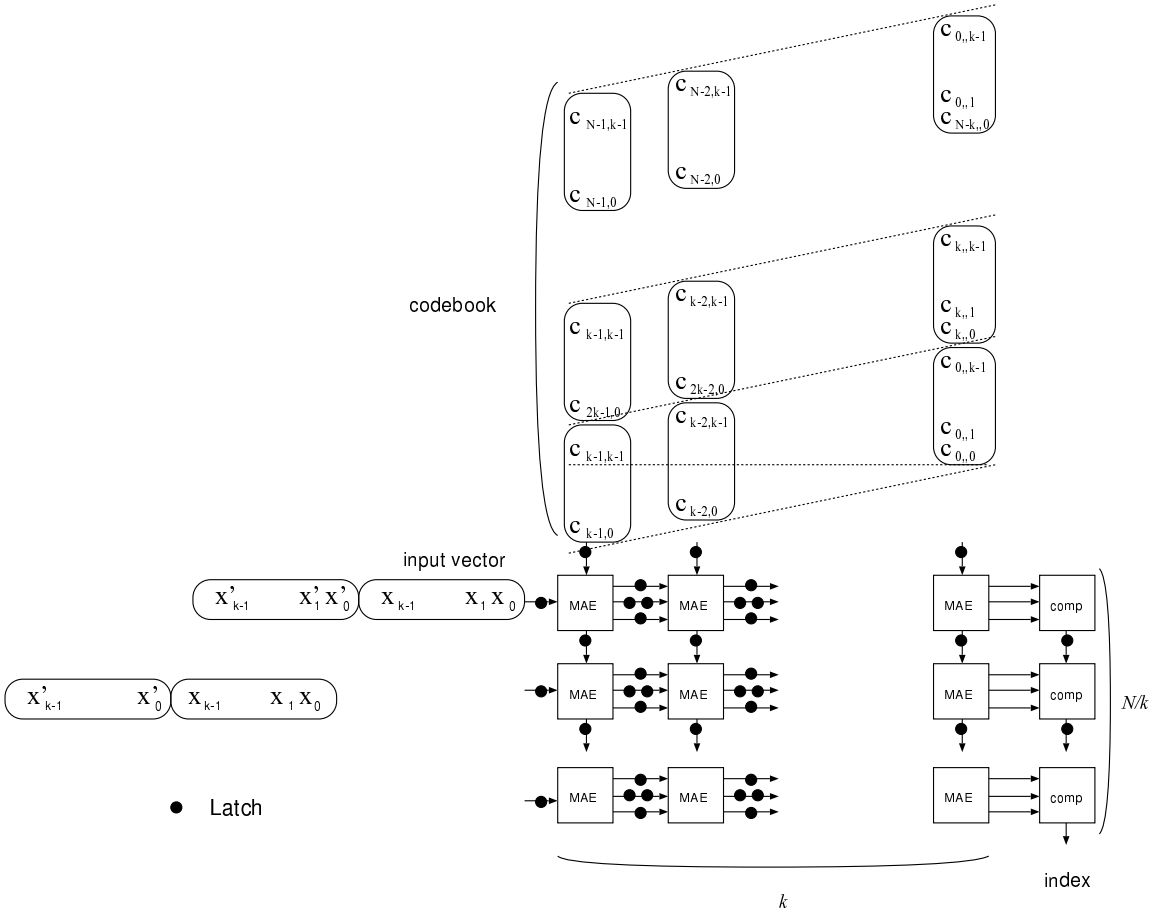


Figure 5.40: A systolic full-searched array processor[WC95].

Reference [CWL96] proposes a serial processor computing the distance using the MSE, which usually requires multiplication. To decrease computation complexity, they split it into additions and table-look-up(TLU) operations. Figure 5.41 shows its block diagram. C_RAM stores a codebook and sends all elements in a code vector $\vec{y}$ to X_UNIT. X_UNIT computes the inner product $\vec{x} \cdot \vec{y}$ using the TLU operation. P_Adder sums up these two terms: $\vec{x} \cdot \vec{y}$ and $|y|^2$. Whole operation can be finished within one clock cycle. It has internal memory and its IO bandwidth is very small. But, the bus width between C_RAM and X_UNIT is quite large (O(km)), which consumes much power.

Reference [SNK+97] describes a 256-element fully-parallel processor as shown in Figure 5.42. The PE cosists of 16 words of SRAM for a code vector and an ALU that computes a MAE and accumulates MAEs. Each PE is laid out into a rectangle region The winner-take-all(WTA) is used to extract the minimum distance. Accumulation and absolute distance computation of a single dimension ($\sum_{j=0}^{i-1} |x_j - y_j| + |x_i - y_i|$) can be obtained in one cycle. The minimum distance is

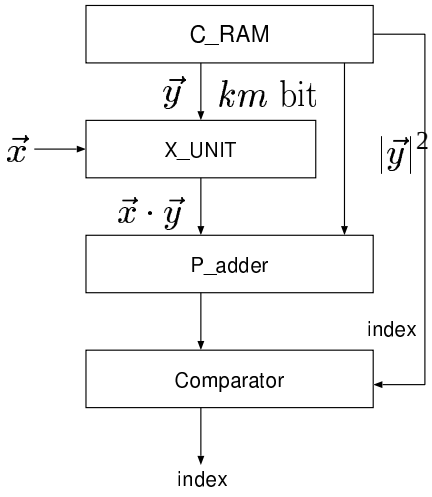


Figure 5.41: A serial full-searched MSE processor[CWL96].

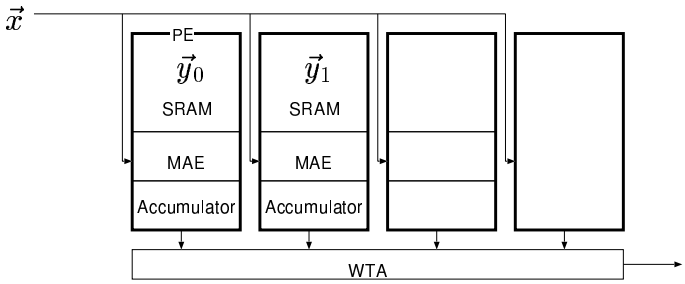


Figure 5.42: A fully-parallel 256-elements parallel processor[SNK+97].

extracted in every 17 cycles. The first 16 cycles is used for 16 iterations of the MAE, and the last cycle is used to extract the minimum distance. It is fastest of all previous implementations. But, it consumes 900mW of power which is too much.

An implementation of Computational RAM as already shown in Figure 2.10 (in page 12) is applied to vector quantization as described in Section 2.2.3. At first, they applied it to still image compression and then to video compression[LP95]. They propose an algorithm to encode 30 CIF (360×288) video frames per second via a low-rate line from 64kbps to 192kbps. The total encoding time is 330ms. on the system including two C∗RAM modules for the nearest neighbor search and an index-based motion estimation (Figure 5.43). A C∗RAM module consists of 4 C∗RAM LSIs. Thus, 40 C∗RAM LSIs are required to compress in real time (33ms.). On the other hand, our proposed algorithm explained in Section 5.8.2 can compress 10 QCIF (176×144) frames per second via a 29.2kbps mobile channel. It requires only one FMPP-VQ64 LSI. Unfortunately, [ESS92] shows only the chip micrograph. Its area or power consumption cannot be seen.

Table 5.15 lists specifications of these 5 implementations, FMPP-VQ64 (VQ64) and FMP-VQ64M (VQ64M). The other implementations are faster than the FMPP-VQs, but the power dissipations of the FMPP-VQs are lowest among all. As for the number of code vectors, almost the other implmentations deal with 256 code vectors, while the FMPP-VQ contains only 64 PEs for 64 code vectors. But the performance and the number of PEs are enough for the current target application, low-rate video compression. The number of PEs can be increased to use the current sub-micron technology. The performance can be improved if the ALU has rich functionalites. The ALU in [SNK+97] is fastest when the input vector is given element by element. But it consumes much power. We have to consider the trade-off between performance and power.

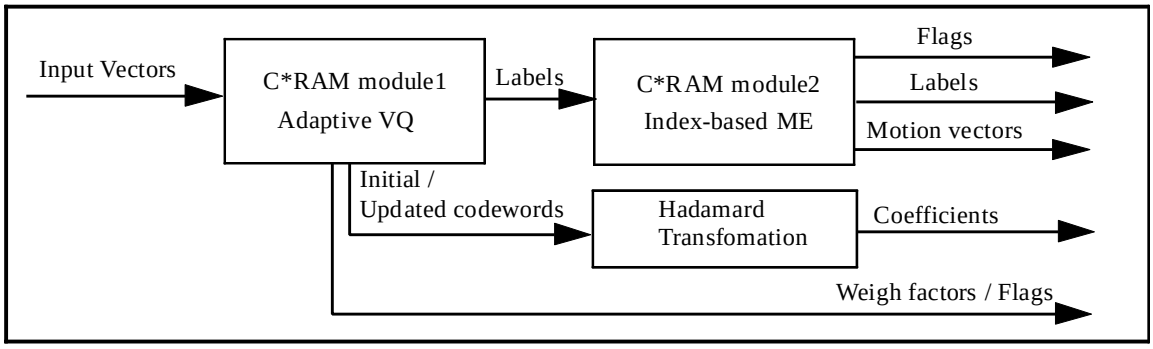


Figure 5.43: C∗RAM implementation of vector quantization for video compression[LP95].

Table 5.15: Comparison with the other vector quantizers.

Name	[FCS+94]	[WC95]	[CWL96]	[SNK+97]	[LP95]	VQ64	VQ64M
measure	MSE	MAE	MSE	MAE	MAE	MAE	MAE
VQ	TSVQ	FSVQ	FSVQ	FSVQ	FSVQ	FSVQ	FSVQ
CV Optimization	N/A	OK	OK	OK	OK	OK	OK
IO Bandwidth	O(m)	O(km)	O(m)	O(m)	O(m)	O(m)	O(m)
Power	500mW	N/A	N/A	900mW	N/A	20mW	10mW
# of CVs	256	256	256	256	64	64	64
Throughput	1.56M	6.25M	195k	2M	N/A	53k	111k
Area(mm^2)	67	42.5	100	72	N/A	52	52
process	1.2μm	0.8μm	1.2μm	0.6μm	N/A	0.7μm	0.7μm
LSI	N/A	N/A	N/A	OK	OK	OK	OK

5.7.2 Comparison with the Von Neumann Sequential Processors.

Here we compare the FMPP-VQ with the Von Neumann commercial sequential processor in terms of speed and power consumption. Table 5.16 lists speed and power of the nearest neighbor search among 64 code vectors for the FMPP-VQ64, the FMPP-VQ64M, Pentium and Ultra SPARC[Pro]. The latter two are commercial sequential CPUs that perform the C program listed in Figure 5.44. The FMPP-VQ64 and the FMPP-VQ64M achieve both of high speed and low power. On the other hand, these commerial CPUs are slower and dissipate more power than the FMPP-VQs.

Figure 5.45 lists the assembler program of a single dimenstion slice of the nearest neighbor search on Ultra SPARC. It consists of two loads, three numerical operations and one conditional branch. From Table 5.16, we can easily guess that it takes about 35nsec.(36.0μs./16/64) to obtain the absolute distance for a single dimension. The four operations besides the first two loads can complete in four

Table 5.16: Speed and power dissipation table of the nearest neighbor search among 64 code vectors.

	Clock Cycle	Bus Cycle	NNS for 64 CVs.	Power	Vdd
FMPP-VQ64	25MHz		18.8μs.	20mW	3.0V
FMPP-VQ64M	25MHz		11.0μs.	(10mW)*	3.0V
Pentium	100MHz	50MHz	85.0μs.	5$\sim$20 W	2.9V
	166MHz	66MHz	49.0μs.		2.9V
Ultra SPARC	300MHz	100MHz	36.0μs.	$\sim$20W	2.6V

$*$ estimated by simulation.

```
        /* Nearest Neighbour Search*/
        #include <stdio.h>
main()
        {
        static int codevector[64][16]={
        /* abbrebiated */
        };
        static int inputvector[16][16]={
        /* abbrebiated */
        };
              int j;
                      int mind=0x1fff,mini=–1;
                      for(j=0;j<64;j++)
                      {
                            int d=0;
                            for(k=0;k<16;k++)
                            {
                                    d+=abs(codevector[j][k]–inputvector[i][k]);
                            }
                            if(d<mind)
                            {
                                    mind=d;
                                    mini=j;
                            }
                      }
        }
```

Figure 5.44: C program for the nearest neighbor search.

clock cycles on the CPU and therefore it takes 20nsec. to load two elements from 100MHz external IO pins. Thus, it is estimated that the assembler program completes in 32nsec., which is almost the same value than the actual processing time 35nsec. Ultra SPARC cannot outperform the FMPP-VQ if the external IO speed remains 100MHz. It takes 20.5μsec. to load 16×64 elements of 64 code vectors.

5.7.3 Comparison with an Application Specific Processor for Vector Quantization

Above these two sections, we compare the FMPP-VQ with the actual implementations: vector quantizers and commercial sequential processors. Here, an application specific sequential processor

```
ld [%o1+%i0],%g3   # load an element of an input vector
ld [%o0+%i1],%g2   # load an element of a code vector
subcc %g3,%g2,%g3# subtract
bneg,a .LL18       # if > 0 goto .LL18
sub %g0,%g3,%g3    # inverse the result
.LL18:
add %i3,%g3,%i3    # accumulation.
```

Figure 5.45: The assembler program to compute the absolute distance of a single dimension.

for VQ is considered to show the limitation of the sequential processing.

Figure 5.46 shows an application specific sequential processor for VQ which consists of an $8 \times 16 \times 64 (= 8k)$ bit SRAM and a processor core. Table 5.17 lists the specification of the SRAM obtained from the 0.7μm process data book[DAT96]. The processor core should access the SRAM 16*64 times for the NNS of 64 code vectors, which takes 13.0μsec. The processor cannnot complete the NNS below 13μsec. As for the power dissipation, the 8kbit SRAM consumses 219mW at 1/12.7ns.(=78.7MHz). Thus, 76,000 NNSs per second is the limitation of this processor. The SRAM consumes the power over 200mW and its area is 2.5mm^2. The FMPP-VQ64 occupies 18mm^2 for the 64 PEs, which is 7.5 times larger than the SRAM. We cannnot estimate the area of the processor core without its circuitry. But, the 8bit micro processor core "Kue-chip2" implemented by a 0.5μm CMOS process occupies 2.37mm^2. The area of the processor including the SRAM and the processor core may be smaller than the FMPP-VQ64, but the power dissipation becomes more than 10 times larger including the SRAM module. It is faster than FMPP-VQ64, but slower than the FMPP-VQ64M. The above processor core accesses code vectors in the SRAM element by element, which eliminates the processing speed. If the multiple SRAM modules are used, the processing speed may improve, but the area and power must be increased.

Table 5.17: SRAM specifications.

	area (mm^2)	access time (ns)	power (mW/MHz)
8bit$\times$1kword SRAM	2.5	12.7	2.79

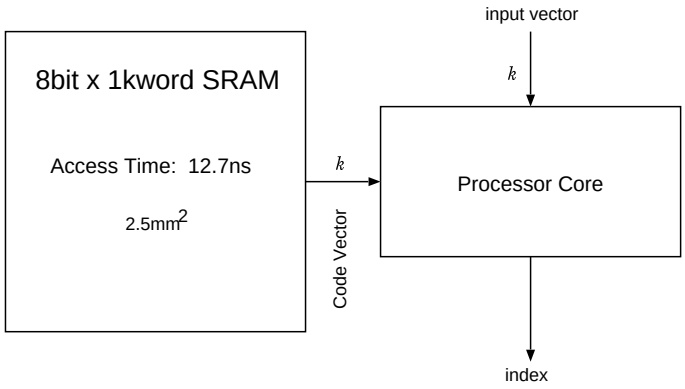


Figure 5.46: An application specific processor for VQ.

5.8 A Low-rate and Low Power Image Compression System Using the FMPP-VQ

In the near future personal digital assistants (PDA) will be a complete voice/video-phone transceiver. Standard video codecs, such as MPEG1, MPEG2, H.261 and H.263, are based on discrete cosine transform (DCT). They consume a large amount of computation on both encoding and decoding, which are not suitable for communication with PDA. On the other hand, Vector Quantization (VQ) has proven to be a powerful technique for low-rate image coding[LBG80]. Compared with DCT-based techniques, a video sequence compressed by VQ can be easily decompressed and has high compression efficiency. On encoding, however, it consumes large computation for the nearest neighbor search (NNS).

Several VQ-based algorithms have been proposed for less computation and high compression ratio. For example, Reference [HH88] has proposed interpolative VQ (IVQ) method, which sends a low resolution interpolated scalar-quantized image while vector-quantizing the residual value. It can reduce blocking effect. Gersho and Shoham suggested hierarchical VQ (HVQ) technique[GS84]. They first introduced a hierarchical structure into VQ-based algorithms. This method partitions large dimensional vectors into small dimensional sub-vectors. HVQ can exploit correlation in large dimensional vectors while avoiding the complexity obstacle of large dimensions. Ho and Gersho proposed multistage hierarchical VQ (MSHVQ)[HG88]. In multistage VQ (MVQ), after an original vector is vector-quantized, the residual vector which has the same dimension as the original one is quantized. MSHVQ technique uses various dimensions at each stage instead of fixed-dimensional vectors. All the above VQ coding schemes were originally proposed for a still image. We present a low-rate video coding algorithm based on MSHVQ. Our algorithm transmits 10 QCIF frames per second via a 29.2kbps mobile wireless channel. It is robust to noise, since indexes from VQ can be coded in a fixed length and a frame of image is always compressed to a fixed size at any video activity.

It enables simple bit rate control by adaptive bit allocation at each stage with small computational complexity.

Here, we introduce a real-time low-rate video compression system using the FMPP-VQ. First, the outline of our video compression system is explained in Section Section 5.8.1. Then, Section 5.8.2 describes the proposed multi-stage hierarchical VQ in detail. The compression system can transmit 10 QCIF frames per second via a 29.2kbps mobile wireless channel. It consists of a PC and a daughter board where the FMPP-VQ is mounted. The detail descriptions of the system and the encoding results from the proposed algorithm are given in Section 5.8.3 and Section 5.8.4 respectively. Simulation results to show robustness to noise are also described in Section 5.8.4.

5.8.1 Overview of the Real-Time Low-Rate Video Compression System

The schematic diagram of our real-time low-rate video compression system is displayed in Figure 5.47. It consists of a host computer and a daughter board where the FMPP-VQ64 is mounted. The compressed data is transmitted via a 29.2kbps channel provided by the PHS terminal. The host computer throws input vectors and receives indexes to/from the daughter board.

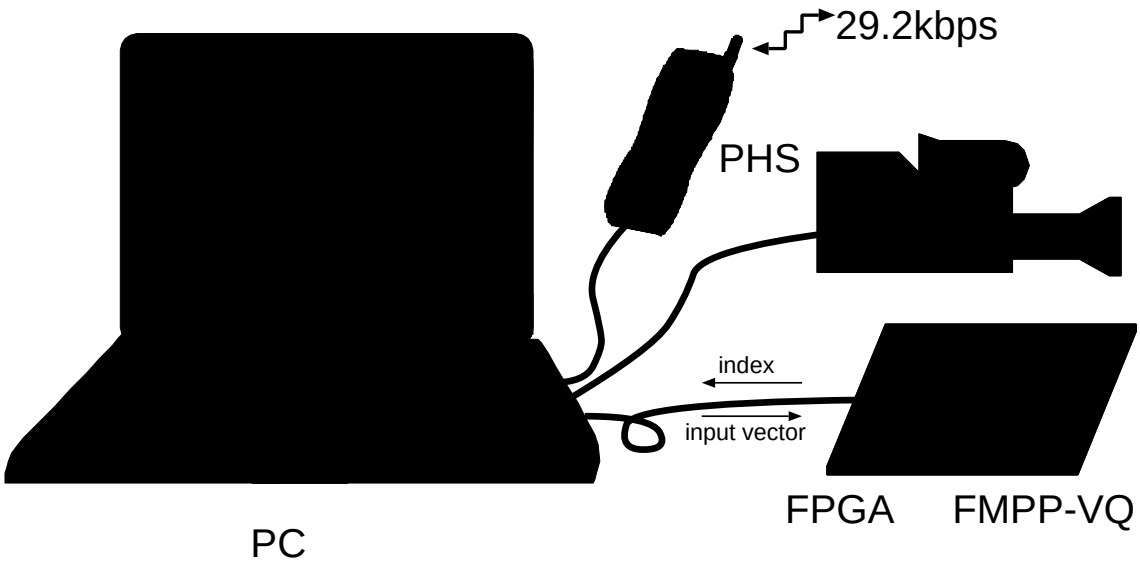


Figure 5.47: Schematic diagram of the low-rate video compression system.

The specifications of our system are as follows.

- Send 176×144 QCIF 8bit gray-scale images at the rate of 10 frames/s via a 29.2kbps wireless channel. Encoding should be done in real time.
- Robust to noise for mobile wireless communication.

- Compression ratio of every frame should be fixed.
- Encoding is performed on a CPU and the FMPP-VQ64, while decoding must be performed on the CPU only.

The most significant specification is the first one. Now we have no real-time mobile videophone terminal. The PHS provides the 29.2kbps digital wireless channel. Robustness to noise is the most important factor for mobile wireless communication. In the DCT-based compression algorithm a pixel block is compressed by a variable length code (VLC). On the other hand, VQ compresses a pixel block to an index, which can be coded by a fixed length. The fixed bit length code is very robust to noise, since the code length can be predicted on the decoder side.

The compression ratio of the current DCT-based video compression algorithm is an average value, which means that compression ratio changes according to video activity. In such condition, the system must have some amount of buffers to store transferring or received data. In our algorithm, a frame of image is compressed to a fixed size. It requires no buffer. It is also robust to noise, since the decoder can easily divide received data to each frame.

5.8.2 Coding Algorithm

Conventional MSHVQ methods[HG88] deal with still image rather than video sequence. We propose the fixed-rate MSHVQ algorithm for real-time low-rate video encoding. In still image encoding, spatial correlation should be used for compression. Compression of video sequence can be done by both temporal and spatial correlations. In our method, VQ compresses spatial correlation, while motion compensation (MC) compresses temporal correlation. MC is first applied to a frame. Then it is hierarchically compressed in multiple stages. The proposed algorithm can adaptively compress video sequence according to video activity. It can transmit 10 QCIF video frames per second via a 29.2kbps transmission line. A QCIF frame is always compressed to 2920bit at any video activity.

Fixed-Rate Multi-Stage Hierarchical VQ

The performance of VQ can be increased according to vector dimensions in order to reduce correlation between input vectors. An inactive area can be partitioned into a large dimensional vector, while an active area must be partitioned into a small dimensional vector. However, a large vector dimension expands computational complexity and memory capacity. We have to prepare a specified quantization method for each different vector dimension. Thus, the vector dimension should be fixed. We adopt a multi-stage method where a frame is hierarchically partitioned according to activity of each area. To fix the vector dimension at 4×4, decimation and interpolation are applied both to enhance the quality and to reduce computational cost.

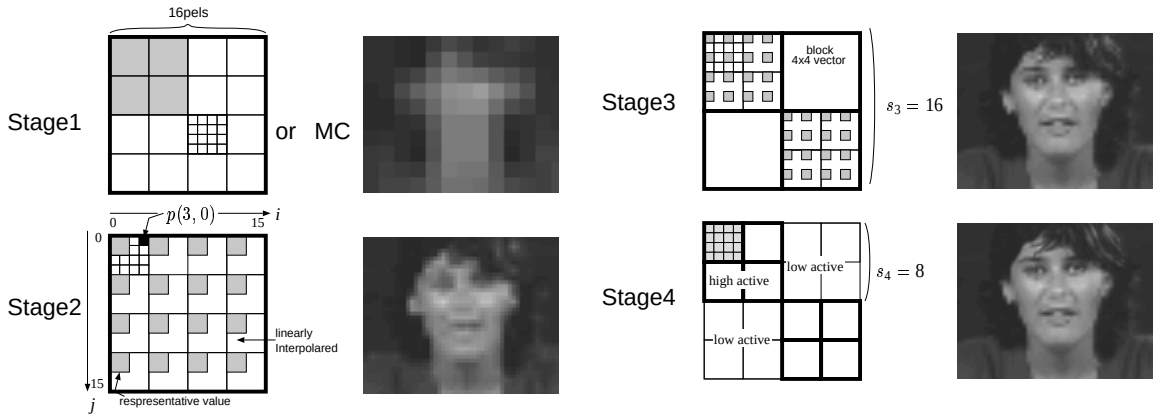


Figure 5.48: Four hierarchical stages for decimation and interpolation.

Images are hierarchically partitioned into blocks in the subsequent four stages (block at every stage is denoted by thick-bordered rectangles in Figure 5.48). Stage 1 scalar-quantizes a value that represents 16×16 pels. At Stage 2, 16 values each of which represents 4×4 pels are vector-quantized. At Stage 3, 16 values each of which represents 2×2 pels are vector-quantized. At Stage 4, a block of 4×4 pels is vector-quantized. We decimate 256 pels into 16 at Stage 1, since an inactive area does not require high resolution. At the rest stages vector dimensions are reduced to 16. Vector dimensions are fixed at 16 all through the stages in order to share the same quantization methodology.

The flow chart of our coding algorithm is depicted in Figure 5.49. From Stage 1 to 3, decimation is done to obtain 16 representative values of a block. There are several possibilities to obtain a representative value. Spatial subsampling causes aliasing errors[HG88]. The mean value of each block brings a blocking effect of a square block. Thus, we use the mean values of the upper-left corner of each block (See Figure 5.48). At Stage 1 the mean value of the upper-left 8×8 pels becomes a representative value. On decoding, empty areas among these representatives are linearly interpolated. This method reduces blocking effect considerably. The decimation schemes at the subsequent stages are equivalent to the above one. At Stage 2, the mean values of the upper left 2×2 pels out of 4×4 pels constitute 16 representative values to be vector-quantized.

At Stage 2 differential values between a decimated original image and a decimated interpolative surface are partitioned into blocks of 16×16 pels. A 4×4 vector is extracted from the 16×16 pels. The blocks with higher activity are chosen to be decoded from Stage 2 to 4. Several blocks compose a macro block to determine activity in order to decrease flags to designate activity. At Stage2, the size of macro block (s_2) is 16×16. At Stage3 and 4, those (s_3 and s_4) are 16×16 and 8×8 respectively. These macro blocks contain 4 blocks. The algorithm to determine the activity is as follows. The differential value $IM(n)$ in Equation (5.16) is computed for every macro block at Stage n.

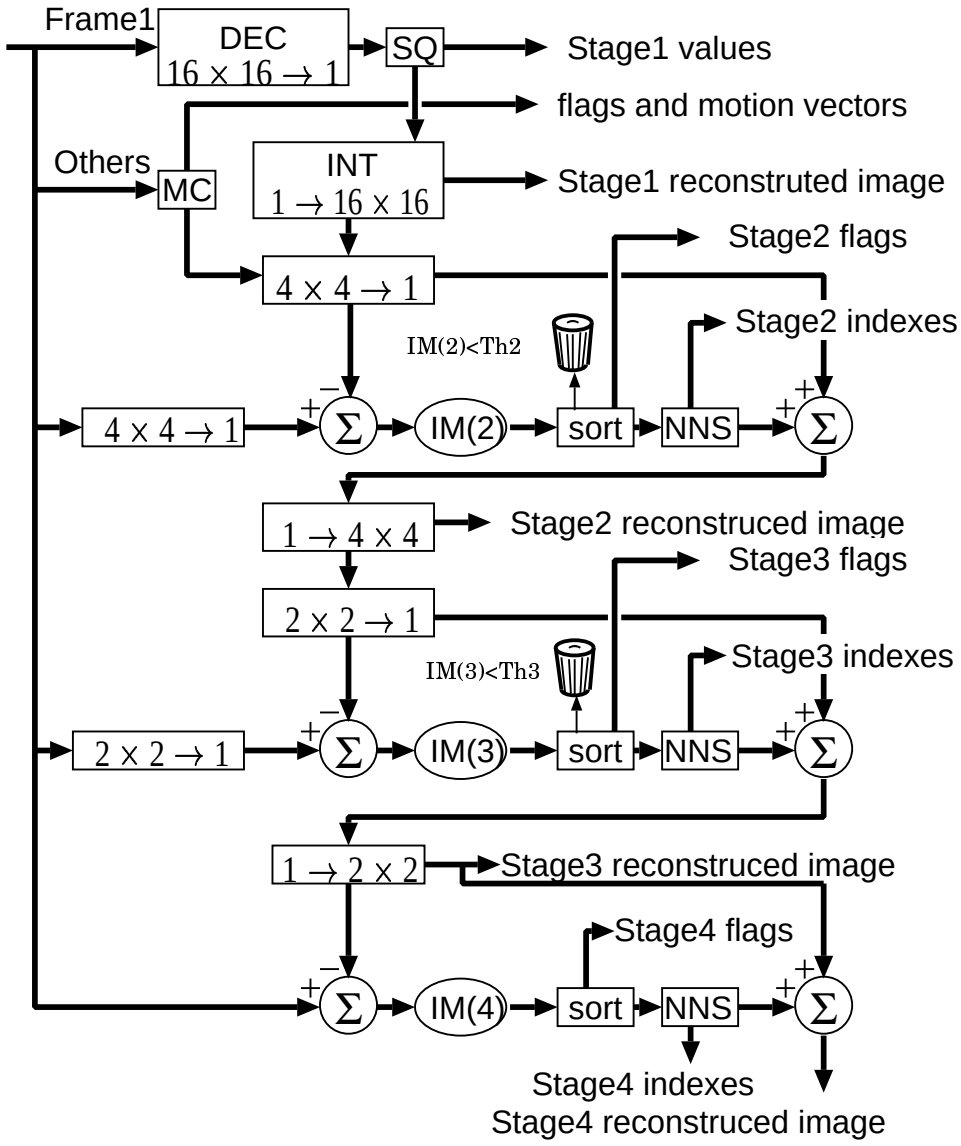


Figure 5.49: Block diagram of our coding algorithm.

$$AD(n-1) = \sum_{i,j=0}^{\sqrt{s_n}} |p_o(i,j) - p_{n-1}(i,j)| \tag{5.14}$$

$$AD(n) = \sum_{i,j=0}^{\sqrt{s_n}} |p_o(i,j) - p_n(i,j)| \tag{5.15}$$

$$IM(n) = AD(n-1) - AD(n) \tag{5.16}$$

$$s_2, s_3 = 16 \times 16, s_4 = 8 \times 8$$

The value $p_n(i,j)$ denotes the (i,j) encoded pixel value of a macro block at Stage n, while p_o is the pixel of the original image (See Figure 5.48). The value $IM(n)$ stands for the improvement of the image quality. All $IM(n)$ values are rearranged in the descending order. The macro block with the

largest $IM(n)$ is given top priority to be vector-quantized. The rest macro blocks are successively vector-quantized until the decoded data reaches to the limit (2920bit). If $IM(n)$ is smaller than a specified threshold value TH_n, the macro block is not transmitted to prevent corruption of image quality by small $IM(n)$ and to leave bits to higher stages. In Figure 5.49, $IM(n)$ is computed before the NNS. It is better to compute $IM(n)$ by the blocks after the NNS. But all differential blocks should be vector-quantized. Thus, $AD(n)$ is computed prior to vector quantization.

At the first frame, Stage 1 sends scalar-quantized values for the lowest resolution. At the subsequent frames, motion compensation (MC) is used instead of Stage 1. In MC, the full search block matching algorithm is most popular, but it requires large computational complexity. We use the orthogonal search method[PHS87], which has good convergence and low computational complexity. A motion vector (MV) is determined for each 16×16-pixel block. The MC search window is 8×8 pels around the center of each block. The following rule determines the motion vector to be transmitted.

1. Compute the following values.

$$AD(x,y) = \sum_{i,j=0}^{15} |p_p(x+i,y+j) - p_o(i,j)| \tag{5.17}$$

$$MV = \min_{-8\le x,y\le 8}{}^{-1} AD(x,y) \tag{5.18}$$

$$p_p : \text{pixel value of the previous frame.}$$

2. The motion vector MV is transmitted if Equation (5.19) is satisfied. It reduces the number of motion vectors to be transmitted.

$$\min AD(x,y) < AD(0,0) + TH_{mc} \tag{5.19}$$

The above hierarchical process are going on until the encoded data reaches the allowable amount (2920bit). If the video activity is high, motion vectors have to compensate temporal activity and lower stages have to produce a large number of indexes to compensate spatial activity. The process is tend to be halted at a lower stage. On the other hand, if the video activity is low, higher stages can produce many indexes to enhance the quality of reconstructed image.

Video coding based on DPCM enlarges the size of transmitted data in high video activity. Two strategies can be chosen when transmitting through a fixed bit-rate. One is to reduce temporal resolution, while the other is to decrease spatial resolution. The latter is better because human eyes are insensitive for spatial activity of the high active video sequence. Lower stages of low spatial resolution are first transmitted in our method. It works conveniently in high activity. When video activity is low, however, lower stages may decrease the quality. It is eliminated by the threshold value TH_n. Our method can offer the way to adapt spatial resolution to video activity.

Codebook Design Strategy

Codebook design strategy is one of the most important factors in VQ. We choose 64 greater values as initial vectors among all the $IM(2)$ values. This is because the reconstructed image at Stage 2 of the first frame seriously affects the quality of subsequent frames. An initial primitive codebook is transmitted at the beginning. It is updated frame by frame.

The FMPP-VQ64 has a capability to vector-quantize an input vector among 64 code vectors. It is not desirable that the size of code vectors is limited to 64. Although a small codebook enlarges distortions, a large codebook increases both the bit width of the index and the size of codebook data. Thus, we generate 1024 code vectors from 64 code vectors to rearrange elements. A primitive code vector turns into 15 derivative code vectors as in Figure 5.50. The bit width of the index increases from 6bit to 10bit, while the size of codebook data is unchanged.

A single code vector is update every frame. We use a modified Linde-Buzo-Gray(LBG) algorithm[LBG80] to update code vectors. The original LBG repeats the sequence until updated code vectors are convergent. On the other hand, our approach applies the LBG once par frame to guarantee real-time encoding. All the updated code vectors are computed on the encoding side. But only a single code vector is transmitted to the decoder side to decrease the data size. It is the vector migrating farthest. Therefore, 16 vectors among 1024 are updated every frame, which reflects the statistical property of the current input frame.

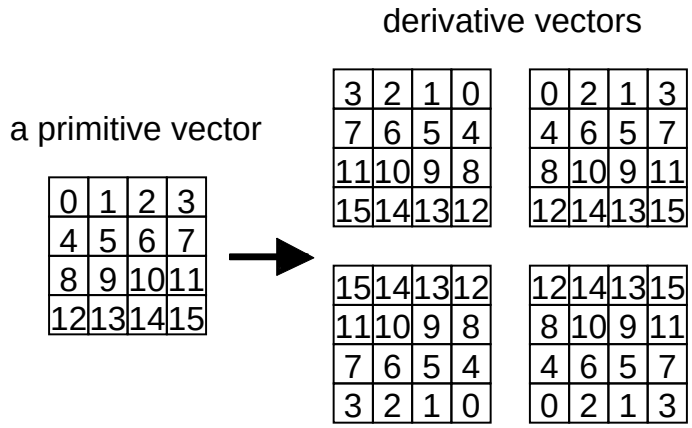


Figure 5.50: Derivative code vectors from a primitive vector.

Coding Strategy to Compensate Errors

Compressed data consist of three parts: flags, motion vectors and vector indexes. Table 5.18 describes contents of compressed data. The vector indexes are very robust to noise, since they are coded in a fixed length. If an error occurs at any index data, the indexes without errors can be correctly

detected. But, if the flags are wrongly transmitted, the decoder mistakes the place of the index, since the number and place of indexes are obtained from the flags. Motion vectors are also important at the decoder side. If the motion vectors are wrongly transmitted, the quality of image decreases considerably. Thus, BCH codes[2] are added to the flags and motion vectors. The BCH codes correct one-bit error and detect three-bit errors. Although the flags are protected by the BCH codes, there is a possibility to lose some flags at the decoder side. If the flag of motion vectors or Stage2 is lost, all the remaining information must be discarded. To protect the remaining information, the number of each flag is also transmitted. The number of flags FM and F2-4 are transmitted twice at the beginning and the end of the frame data to securely protect them from noise. If an error occurs, there happens a conflict between the number of flags derived from FM and F2-4 and the that from NM and N2-4. If the difference between them is only 1 at Stage$n$, the indexes are decoded according to Nn. If the difference is larger than 1, the indexes of Stage$n$ are discarded and the remaining information is decoded at the subsequent stages.

In H.263, compressed rate varies according to the encoding data. If an error occurs, it is difficult to re-synchronize data without some extra flags for synchronization. In the proposed algorithm, the encoder can easily synchronize data at every frame, since a frame is decoded in a fixed size of 2920bit.

5.8.3 Experimental Real-Time Low-Rate Video Compression System

We develop an experimental real-time low-rate video compression system composed of a PC, an FPGA and an FMPP-VQ64 LSI. The latter two LSIs are mounted on a daughter board. Figure 5.51 shows the experimental system. The CPU (Pentium 200MHz) on the PC performs the proposed MSHVQ algorithm except for the NNS on the FMPP-VQ64 controlled by the FPGA.

A VQ index is 10bit long for 1024 code vectors derived from 64 primitive vectors. First, the FMPP-VQ64 generates an index of the nearest code vector among the 64 primitive vectors. Then it accepts 15 rearranged derivative input vectors to generate indexes for them. An input vector is rearranged instead of code vectors. A 10bit index is extracted for each block. A frame should be encoded below 2920 bit within 100ms. At most, 2920/10(=292) VQ indexes should be computed for a single frame. Thus, the FMPP-VQ must perform $2920/10 \times 16$ (=4,672) NNSs per frame, which is within its capability of 53,000 NNSs per second. The number of NNSs obtained from the actual compression stream is 2540 per frame which takes 48ms. The compression of a frame except the NNS takes 30ms on Pentium 200MHz. Thus, the system can perform the compression in 78ms even when the NNS and the other operations are done in serial. If the performance of the CPU is poorer, the NNS and the other operations can be done in parallel. Note that the NNS for 64 code vectors takes 80ns on Pentium 200MHz, while that takes 18.8ns on the FMPP-VQ64. The proposed algorithm

[2] Bose Chaudhuri-Hocquenghem code.

Table 5.18: Contents of compressed 2920bit data.

Contents	# of bits	BCH code
flags for motion vectors (FM)†	99	7
Stage2 flags ($F2$)	99	7
Stage3 flags ($F3$)	99	7
Stage4 flags ($F4$)	0 or 396	9
# of motion vectors (NM)†	7	10
# of Stage2 flags ($N2$)	7	
# of Stage3 flags ($N3$)	7	
# of Stage4 flags ($N4$)	9	
indexes of Stage1 ($I1$)*	8×99	0
motion vectors (MV)†	$8 \times NM$	30
indexes of Stage2 ($I2$)	$10 \times N2$	0
indexes of Stage3 ($I3$)	$10 \times 4 \times N3$	0
indexes of Stage4 ($I4$)	$10 \times 4 \times N4$	0
updated code vector	128	16
code vector index	6	
padding data	2920−#total	-

$*$ At the 1st frame. † From the 2nd frame.

cannot be done in real-time without the FMPP-VQ64.

The decompression procedure of a single frame needs only 3.8ms on Pentium 200MHz without using the FMPP-VQ64, while the decompression of H.263 takes 9.8ms. The compressed data from the proposed algorithm can be decoded almost 2.5 times faster than that of H.263. Note that both decoding programs are optimized to the same level and no MMX code is used. Figure 5.52 shows computation amount of each function of the proposed fixed-rate MSHVQ and H.263. The function "Stage2-Stage4" of the proposed replaces the indexes to code vectors according to the flags, which function is done by a simple table look-up method. On the other hand, H.263 should perform the complex IDCT and VLC. Thus the proposed algorithm can decode the compressed data much faster than H.263.

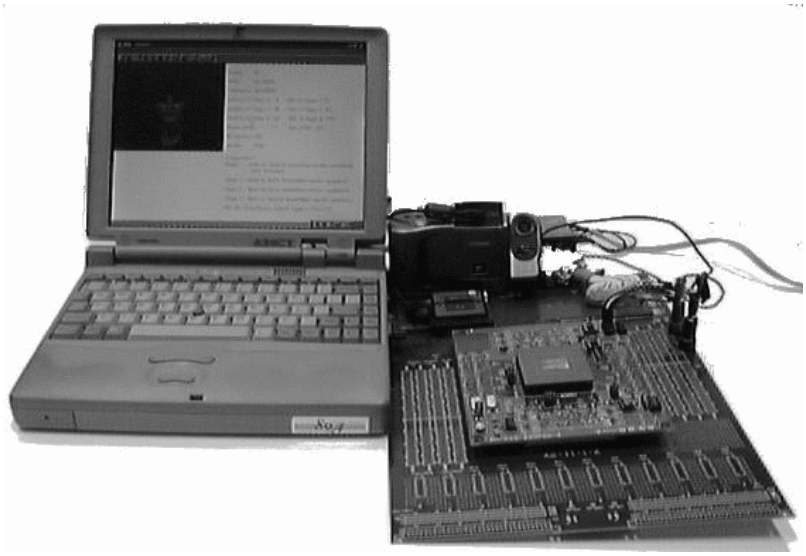


Figure 5.51: Experimental real-time low-rate video compression system.

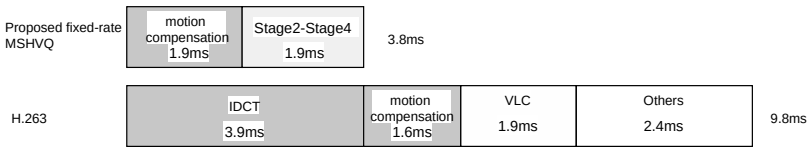


Figure 5.52: Computation amount of each function on decoding of the fixed-rate MSHVQ and H.263.

5.8.4 Performance Evaluation

Here, we show several simulation results of the proposed fixed-rate MSHVQ algorithm.

Quality of Decoded Images

We apply the proposed fixed-rate MSHVQ algorithm to the standard video sequences. Figure 5.53 shows the temporal PSNR transitions of the proposed algorithm and H.263[3] for "Suzie." Both are going to send 10 QCIF frames over a 29.2kbps line. During the first three frames of our algorithm, initial 64 code vectors are transmitted prior to the compressed data. Thus, encoding starts from the fourth frame (0.4s).

Figure 5.54 shows temporal bit allocation at each stage. In our adaptive method, every frame is compressed in a fixed size of 2920bit. Suzie shakes her head near 1.8 second, when the motion is most active. The proposed algorithm allocates motion vectors and the indexes of Stage 2 to most of bits at that time. The results indicate that transmitted bits are properly assigned for each stage. In

[3]ITU-T SG15 Experts Group on Very Low Bitrate Visual Telephony: "Video Codecs Test Model, TMN5," (1995). No option is used.

Figure 5.54, the temporal bit allocation of H.263 dynamically fluctuates according to the temporal and spatial activity. At the 1.8 second, H.263 skips a frame so as not to exceed the limit (29.2kbit/s), since the compressed frame amounts to over 5000 bit. Table 5.19 shows the average PSNRs for 8 standard video sequences. The column "1024 CVs" shows PSNRs of the proposed fixed-rate MSHVQ algorithm. The quality of reconstructed image derived from the algorithm is only 2.5dB worse on the average compared with H.263.

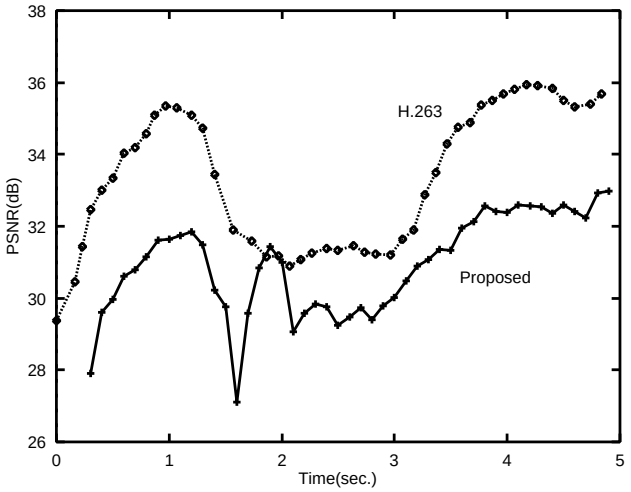


Figure 5.53: PSNRs of the proposed MSHVQ algorithm and H.263 for "Suzie."

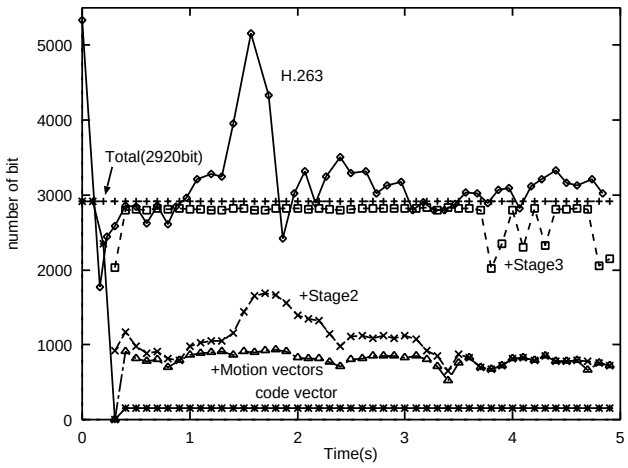


Figure 5.54: Temporal bit allocation of Suzie for the proposed MSHVQ algorithm and H.263.

We evaluate the effect of the derivative code vectors from 64 code vectors. The column "64 CVs" in Table 5.19 shows the PSNRs obtained from 64 code vectors. The proposed method using 1024 code vectors always enhances the PSNRs by an average of 1.1dB, even though the bit width of the index increases from 6bit to 10bit.

Robustness to Noise

To evaluate the robustness to the noise, two error conditions from MPEG4 error-resilience test conditions[MPE95] in Table 5.20 are applied to Mother&Daughter (Mot&Dau), Miss America and Suzie. Note that we modify Multiple Burst Errors condition to "2 burst errors in [1.5,5]" for Suzie and Miss America, since they are 5 second long.

Table 5.21 shows the average PSNRs from High Random BER (HRB) and Multiple Burst Error (MBE) conditions. Figure 5.55 and Figure 5.56 depict the temporal PSNR transitions from HRB and MBE using Mother&Daughter respectively. The topmost line shows the original no-error condition.

Reference [MN96] has proposed a self-synchronized coding scheme for H.263, which shows the PSNR transitions in the condition of 24kbps and 48kbps. The PSNR curves drop drastically from 30dB to 15 or 20dB on the 10^{-3} random BER condition. Although simulation conditions are different, it is evident that H.263 is extremely weak to noise because of the VLC and the activity-

Table 5.19: Average PSNRs for 8 standard video sequences.

	PSNR (dB)		
	Proposed MSHVQ		H.263
Sequence Name	64 CVs	1024 CVs	
Miss America	35.4	36.4	39.7
Grandmother	32.2	33.7	35.6
Suzie	30.2	30.9	33.4
Mother&Daughter	29.3	30.7	33.4
Salesman	27.7	29.9	32.3
Trevor	27.0	27.9	30.5
Carphone	26.4	27.2	29.8
Foreman	25.3	26.0	28.3
average	29.2	30.3	32.8

64 CVs: Use only primitive vectors.

1024 CVs: Use derivative vectors.

H.263: No option is used.

oriented compression ratio as already shown in Figure 5.54. Our simulation results show that the proposed fixed-rate MSHVQ algorithm is very robust to both random and burst errors. But, we have to perform more simulations to fairly compare the error robustness for both algorithm. The PSNR drops of our fixed-rate MSHVQ algorithm are always very small values of 1 or 2 dB. The PSNR drop of the MBE condition from Suzie is largest of all (2.3dB), since the compressed data contains lots of motion vectors. If the motion vectors are lost by noise, the quality of reconstructed image seriously decreases.

Table 5.20: Error conditions[MPE95].

Residual error conditions	Description	Error interval [begin,end(s)]
10^{-3} Random Bit Error Rate	High Random BER (HRB)	[1.5,end]
3 burst of errors 50% BER within burst Random Burst Length: 16 to 24 ms Random bursts separation: $>$ 2s	Multiple Burst Errors (MBE)	[1.5,8]

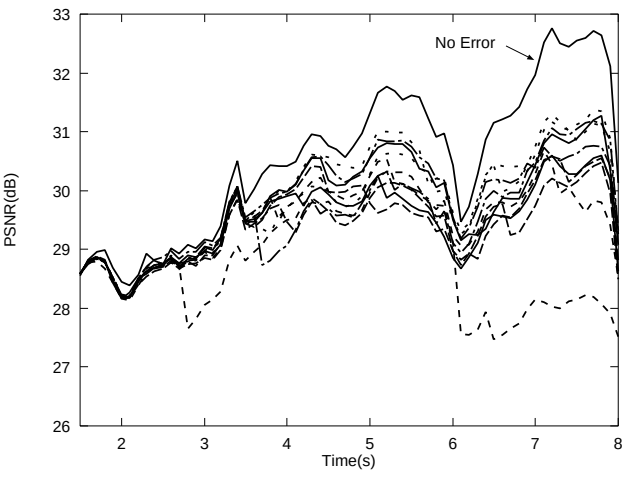


Figure 5.55: PSNR transitions from High Random BER (10^{-3}) using Mother&Daughter.

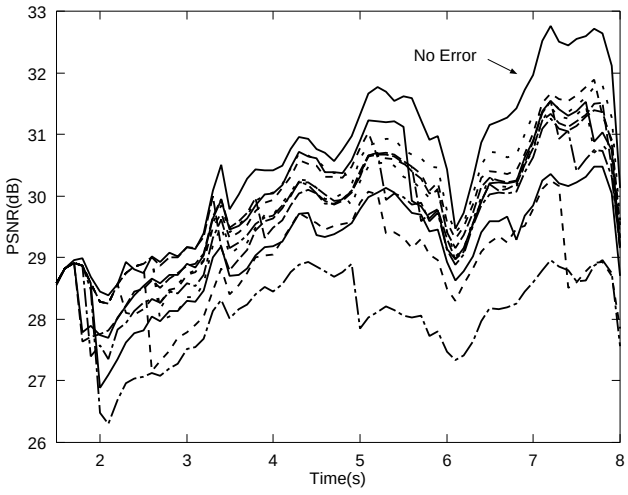


Figure 5.56: PSNR transitions from Multiple Burst Errors using Mother&Daughter.

Table 5.21: Average PSNRs from High Random BER (HRB) and Multiple Burst Errors (MBE) conditions.

	Average PSNR(dB)					
Sequence	Mot&Dau		Miss America		Suzie	
No Error	30.5		36.4		31.0	
Random Seeds	HRB	MBE	HRB	MBE	HRB	MBE
10	29.9	30.1	35.9	36.0	30.7	30.6
11	28.7	29.9	35.1	35.9	29.2	27.5
12	29.4	29.8	35.9	36.1	30.2	30.1
13	29.8	29.6	36.1	36.2	30.5	30.1
14	29.5	29.9	35.8	34.9	30.5	25.8
15	29.8	29.8	35.5	35.3	30.5	28.4
16	29.5	28.9	36.1	34.5	30.0	27.6
17	29.4	29.8	36.1	35.5	30.7	30.7
18	29.7	28.1	34.5	35.0	30.0	26.4
19	29.8	29.1	35.7	35.9	30.6	30.1
Average	29.5	29.5	35.7	35.5	30.3	28.7
PSNR drop	-1.0	-1.0	-0.7	-0.9	-0.7	-2.3

5.9 Summary of the Chapter

In this chapter, an implementation of the FMPP for vector quantization (FMPP-VQ) is described in detail. Vector quantization (VQ) is very much applicable to the memory-based architecture. In VQ a single input vector is compared with lots of code vectors. The output value is only an index of the nearest vector. We can obtain indexes for multiple input vectors without changing the content of code vectors. An input vector can be broadcast through the shared bus.

We have designed and fabricated 3 LSIs. The first one integrates 4 PEs in order to verify its functionalities, which is almost fully functional at 25MHz. The other two LSIs contains 64 PEs to be applied for actual image compression. The FMPP-VQ64 integrates 64 PEs and a simple control logic, which is fully functional besides the control logic. It can perform 53,000 nearest neighbor searches (NNSs) per second, while its power consumption is only 20mW. The FMPP-VQ64M integrates 64 reorganized PEs and a highly-functional control logic. The strategy to compute the absolute distance is optimized. Therefore, the performance is doubled, while its power dissipation is reduced to half. The highly-functional control logic automatically proceeds the procedure for the NNS. Memory-based parallel processing allows both of high performance and low power. The commercial micro processor working at 300MHz cannot exceed the performance of the FMPP-VQ, while dissipating more power.

We have also developed a real-time low-rate video compression system based on VQ. It can transmit 10 QCIF (176×144) video frames per second through a 29.2kbps wireless line. Our developed compression algorithm uses an adaptive multi-stage hierarchical vector quantization. When video activity is high, large pixel blocks for low resolution are mainly transmitted. When it is low, small pixel blocks for high resolution increase the quality. It is robust to noise, since the fixed length code is used instead of the variable length code as in the DCT-based algorithms and every frame is compressed to a fixed size. It achieves PSNRs over 30dB for the well-known standard video sequences. It is robust to both random and burst errors. The PSNR drops are only 1 or 2dB. The proposed algorithm can encode the QCIF video sequence in real-time on a CPU and an FMPP-VQ64 LSI. The decoding process can be done on the CPU only. We are going to develop the experimental encoding system which consists of a personal computer and a daughter board with an FMPP-VQ64 LSI. The most intensive future work of this research is to develop a portable videophone system for mobile communication. It will be realized by the FMPP-VQ64M LSI and an application specific LSI for encoding and decoding.

Chapter 6

Conclusion

In this paper, a memory-based SIMD shared-bus parallel processor, “Functional Memory Type Parallel Processor (FMPP)” is discussed. Current computers consist of a fast CPU (processor) and slow DRAM modules. Between them, there is a shared bus where all the data and codes (programs) should be passed through. The shared bus causes so-called Von Neumann bottleneck, where the system performance is limited by the performance of the bus. The FMPP architecture integrates a memory and a processor on a single LSI. These two components, a memory and a processor, are closely linked in the FMPP. A processing element (PE) contains some amount of memory and an ALU. All PEs are connected thorough a shared bus and laid out in a two-dimensional regular array structure. The FMPP has a capability to break the bottleneck. The FMPP-based computing system, where the part of main memory is replaced with the FMPP, shows better performance than the conventional Neumann computer. For example, an FMPP-based system with 1000 PEs can perform SIMD operations 40 times faster than a conventional Von Neumann computer.

The bit-parallel block-parallel structure is proposed and discussed here. A PE of the BPBP structure consists of several words and a bit-parallel ALU. It is suitable for arithmetic computations such as addition or multiplication. We have developed and fabricated a 1kbit BPBP-FMPP LSI, which is the first prototype FMPP LSI and works as a RAM, a CAM and a parallel processor. The capability of addition in a bit-parallel manner enables arithmetic computations inside memory storage cells without transferring data between a memory and a CPU. We propose a new strategy for addition using a CAM-based memory and a Manchester carry chain without using any conventional adder. The LSI has various functionalities of numerical and logical operations. The BPBP-FMPP can be applied for the knapsack problem, one of NP-hard combinatorial optimization problems. The BPBP-FMPP is 100,000 times faster than the sequential implementation when the number of luggage is 20.

We have proposed an application specific bit-parallel block-parallel FMPP for Vector Quantization (FMPP-VQ) to accelerate the nearest neighbor search (NNS) of vector quantization. Each processing element computes the distance between an input vector and a code vector and finally the code vector nearest to the input vector can be obtained rapidly using the CAM-based parallel search. Memory-

based architecture and the ALU using pass-transistor logic minimize circuit area considerably. An LSI including four PEs has been implemented in a 0.7μm CMOS process. It operates at 25MHz. Then, we have developed the FMPP-VQ64 containing 64 PEs, which is fabricated using the same 0.7μm CMOS process. It performs 53,000 NNSs for 16-dimensional code vectors. The power consumption is 20mW at the condition of 25MHz clock frequency and 3.0V power supply voltage. The modified version of the FMPP-VQ64 called FMPP-VQ64M has been fabricated. It performs 111,000 NNSs per second, while its power consumption is estimated to 10mW. It also integrates highly-functional control logic. The memory-based architecture enables both of high performance and low power. A serial implementation including 8kbit SRAM for code vectors has a capability to achieve almost the same performance than the FMPP-VQ LSIs, but it must dissipate 10 times larger power than the FMPP-VQ.

We have proposed a hierarchical multi-stage vector quantization algorithm for real-time low-rate video compression using the FMPP-VQ. It can transfer 10 QCIF frames per second over a 29.2kbps mobile wireless channel. It is robust to noise, since a pixel block is compressed to an index coded with a fixed length. A frame of image is first motion-compensated. Then the residual surface is hierarchically compressed by multiple stages. The vector dimension is fixed to 16 (4×4) all through the stages to share the same quantization methodology. Large blocks over 4×4 pixels are decimated to 16. The algorithm compresses video frames adaptively to the activity. For high-active frames, motion vectors and large blocks for low resolution are mainly transmitted to compensate temporal activity. For low-active frames, small blocks enhance the image quality. It achieves the PSNR over 30dB for the well-known standard video sequences. The quality of reconstructed image is only 2.3dB worse than that from H.263.

The proposed algorithm is done in real-time on the experimental video compression system composed of a PC and the FMPP-VQ64 for the nearest neighbor search. Compressed data from the proposed algorithm can be easily decoded. It takes 4.4ms. on Pentium 200MHz, while H.263 takes 10.9ms. Our future task is to build a portable real-time low-rate videophone system for mobile field. The proposed algorithm will be done on a low-power application specific LSI and the FMPP-VQ64 LSI.

The FMPP allows massively parallel processing inside memory. But the actual LSI implementations do not have enough parallelism. The fist implementation 1kbit BPBP-FMPP has only 8 PEs, which is mainly because its high functionality and 32bit operation capability. The fabricated process is also the 1.2μm CMOS process, which is old-fashioned. In the FMPP-VQ architecture, we have implemented 64 PEs using the 0.7μm CMOS process. The functionality and bit-width of the PE is eliminated for vector quantization, which enhance the integration density. These 64 PEs are enough for vector quantization. But, the FMPP cannot show massively parallel computation on such an

application specific implementation. The area of the FMPP-VQ64 including 8kbit codebook words is 10 times bigger than the conventional 8kbit SRAMs. Although the FMPP architecture allows high integration density to eliminate communication between PEs and to control all the PEs with the same instructions, the bus and control lines still occupy large area. The brand-new fine grain sub-micron process, however, will allow huge number of processors on a single die.

In this paper, two bit-parallel block-parallel FMPPs are proposed. The BPBP-FMPP is for general purpose and the FMPP-VQ is application-specific. At first, the FMPP architecture is proposed to be used as a part of main memory for general purpose processing. The current research, however, tends to aim a specific application such as vector quantization. As discussed in Chapter 3, the FMPP-based computing system outperforms the conventional Von Neumann computers. Our future task is to develop an FMPP-based computing system including the following components:

- An FMPP LSI for a part of main memory,
- A processor that can handle the memory-based processing inside the FMPP.
- A compiler that can assign SIMD operations to the FMPP.

The block-parallel structure is suitable for such a system, since huge number of numerical operations among multiple words can simultaneously be done in bit-parallel.

Bibliography

[CSB92] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen. Low-power CMOS digital design. *IEEE J. Solid-State Circuits (USA)*, 27(4):473–84, 1992.

[CWL96] H.Q. Cao, C.C. Wang, and W. Li. Variable rate lattice VQ algorithm for vector subband coding. *Proc. SPIE - Int. Soc. Opt. Eng. (USA)*, 2727(pt.1):319–30, 1996.

[DAT96] *ES2 ECPD07 Library Databook*. ATMEL ES2, 1996.

[ERA] *http://www.mitsubishi-chips.com/eram/whatis.htm*.

[ESS92] D.G. Elliott, W.M. Snelgrove, and M. Stumm. Computational RAM: a memory-SIMD hybrid and its application to DSP. *Proceedings of the IEEE 1992 Custom Integrated Circuits*, pages 30.6/1–4, 1992.

[FCS+94] Wai-Chi Fang, Chi-Yung Chang, B.J. Sheu, O.T.-C. Chen, and J.C. Curlander. VLSI systolic binary tree-searched vector quantizer for image compression. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst. (USA)*, 2(1):33–44, 1994.

[FOT93] Y. Fujino, T. Ogura, and T. Tsuchiya. Facial image tracking system architecture utilizing real-time labeling (TV telephones and conferencing). *Proc. SPIE - Int. Soc. Opt. Eng. (USA)*, 2094(pt.1):2–11, 1993.

[Fro98] R. Fromm. IRAM(Intelligent RAM) Solution for Merged DRAM/Logic LSI Architecture. *ASP-DAC'98 Tutorial 1: Merging DRAM and Logic Possibilities and Challenges*, pages 63–90, 1998.

[FYO92] Y. Fujita, N. Yamashita, and S. Okazaki. IMAP: integrated memory array processor. *J. Circuits Syst. Comput. (Singapore)*, 2(3):227–45, 1992.

[GC83] A. Gersho and V. Cuperman. Vector Quantization. *IEEE Commun. Mag.*, 21(9):15–21, 1983.

[GG92] A. Gersho and R.M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Boston, 1992.

[GS84] A. Gersho and Y. Shoham. Hierarchical vector quantization of speech with dynamic codebook allocation. *ICASSP 84. Proceedings of the IEEE International Conference on*, pages 10.9/1–4 vol.1, 1984.

[GS97] J.C. Gealow and C.G. Sodini. A Pixel-Parallel Image Processor Using Logic Pitch-Matched to Dynamic Memory. *Symposium on VLSI circuits*, pages 57–58, 1997.

[HAK97] T. Hanyu, M. Arakaki, and M. Kameyama. Design and evaluation of a 4-valued universal-literal CAM for cellular logic image processing. *IEICE Trans. Electron. (Japan)*, E80-C(7):948–55, 1997.

[HG88] Y.-S. Ho and A. Gersho. Variable-rate multi-stage vector quantization for image coding. *ICASSP 88: 1988 International Conference on Acoustics, Speech,*, pages 1156–9 vol.2, 1988.

[HH88] H. Hang and B. Haskell. Interpolative Vector Quantization of Color Images. *IEEE Trans. on Comm.*, COM-36:465–470, 1988.

[Hil87] W.D. Hillis. *The Connection machine*. MIT Press, 1987.

[HK96] M. Hariyama and M. Kameyama. Collision detection VLSI processor for intelligent vehicles based on a ROM-type content-addressable memory. *Trans. Inst. Electron. Inf. Commun. Eng. C-II (Japan)*, J79C-II(11):698–705, 1996.

[HS92] F.P. Herrmann and C.G. Sodini. A dynamic associative processor for machine vision applications. *IEEE Micro (USA)*, 12(3):31–41, 1992.

[HS95] F.P. Herrmann and C.G. Sodini. A 256-element associative parallel processor. *IEEE J. Solid-State Circuits (USA)*, 30(4):365–70, 1995.

[INK+95] K. Inoue, H. Nakamura, H. Kawai, T. Tani, Y. Sakemi, H. Matsuoka, M. Ishikawa, J. Matsumoto, K. Yamamoto, K. Takahashi, M. Yamawaki, E. Yokomoto, C.A. Hart, J. Lin, K. Ishihara, and K. Shimotori. A 10 Mb 3D frame buffer memory with Z-compare and alpha-blend units. *1995 IEEE International Solid-State Circuits Conference. Digest*, pages 302–3, 384, 1995.

[KKT+96] K. Kobayashi, M. Kinoshita, M. Takeuchi, H. Onodera, and K. Tamaru. A memory-based parallel processor for vector quantization. *22nd European Solid-State Circuits Conference*, pages 184–187, 1996.

[KKT+97] K. Kobayashi, M. Kinoshita, M. Takeuchi, H. Onodera, and K. Tamaru. A memory-based parallel processor for vector quantization: FMPP-VQ. *IEICE Trans. on Electron*, E80-C(7):970–975, 1997.

[KNA+95] T. Kimura, K. Nakamura, Y. Aimoto, T. Manabe, N. Yamashita, Y. Fujita, S. Okazaki, and M. Yamashina. Design of 1.28-GB/s high bandwidth 2-Mb SRAM for integrated memory array processor applications. *IEEE J. Solid-State Circuits (USA)*, 30(6):637–43, 1995.

[KNK+92] Y. Kuwahara, K. Nakamura, K. Kubota, M. Sato, and T. Ohtsuki. CAM-based Hardware Engine for Geometrical Problems. *CPSY92-17*, pages 63–70, 1992.

[KNT+98] K. Kobayashi, N. Nakamura, K. Terada, H. Onodera, and K. Tamaru. An LSI for Low Bit-Rate Image Compression Using Vector Quantization. *IEICE Trans. on Electron*, E81-C(5):718–724, 1998.

[Koh87] T. Kohonen. *Self-Organization and Associative Memory*. Springer-Verlag, Berlin Heidelberg, 1987.

[KOT95] K. Kobayashi, H. Onodera, and K. Tamaru. A bit-parallel block-parallel functional memory type parallel processor LSI for fast addition and Multiplication. *Symposium on VLSI circuits*, pages 61–62, 1995.

[KTYO93] K. Kobayashi, K. Tamaru, H. Yasuura, and H. Onodera. A bit-parallel block-parallel functional memory type parallel processor architecture. *IEICE Trans. on Electron*, vo.E76-C(7):1151–1158, 1993.

[LBG80] Y. Linde, A. Buzo, and R.M. Gray. An Algorithm for Vector Quantizer Design. *IEEE Trans. Commun. (USA)*, COM-28(1):84–95, 1980.

[LP95] T.M. Le and S. Panchanathan. Computational RAM implementation of an adaptive vector quantization algorithm for video compression. *IEEE Trans. Consum. Electron. (USA)*, 41(3):738–47, 1995.

[MN96] Y. Matsumura and T. Nakai. Self-Synchronized Syntax for Error-Resilient Video Coding. *IEICE Trans. Commun. (Japan)*, E79-B(10):1467–73, 1996.

[MPE95] MPEG-4 Testing and Evaluation Procedures Document. pages 32–34, 1995.

[NEC] *http://www.incx.nec.co.jp/imap-vision/*.

[NO90] J. Naganuma and T. Ogura. CAM-based Prolog machine and its performance evaluation. *Trans. Inst. Electron. Inf. Commun. Eng. D-I (Japan)*, J73D-I(11):856–63, 1990.

[NYT89] A. Nakano, H. Yasuura, and K. Tamaru. Functional memory type parallel architecture for image processing. *Proc. of 1989 IEEE Int. Conf. on VLSI*, pages 329–328, 1989.

[ON97] T. Ogura and M. Nakanishi. CAM-based highly-parallel image processing hardware. *IEICE Trans. Electron. (Japan)*, E80-C(7):868–74, 1997.

[ONB+96] T. Ogura, M. Nakanishi, T. Baba, Y. Nakabayashi, and R. Kasai. A 336-kbit content addressable memory for highly parallel image processing. *Proceedings of the IEEE 1996 Custom Integrated Circuits*, 2094(pt.1):273–6, 1996.

[OYN85] T. Ogura, S. Yamada, and T. Nikaido. A 4kb associative memory LSI. *IEEE J. Solid-State Circuits*, SC-20(6):1277–1282, 1985.

[OYY86] T. Ogura, S. Yamada, and J. Yamada. A 20 Kb CMOS associative memory LSI for artificial intelligence machines. *Proceedings of the IEEE International Conference on Computer*, pages 574–7, 1986.

[PHS87] A. Puri, H.M. Hang, and D.L. Schilling. An Efficient Block Matching Algorithm for Motion-Compensated Coding. *ICASSP*, pages 1063–1066, 1987.

[Pro] *http://infopad.eecs.berkeley.edu/CIC/summary/local/*.

[SKO90] M. Sato, K. Kubota, and T. Ohtsuki. A hardware implementation of gridless routing based on content addressable memory. *27th ACM/IEEE Design Automation Conference. Proceedings 1990*, pages 646–9, 1990.

[SNK+97] T. Shibata, A. Nakada, M. Konda, T. Morimoto, T. Ohmi, H. Akutsu, A. Kawamura, and K. Marumoto. A Fully Parallel Vector Quantization Processor for Real-Time Motion Picture Compression. *1997 IEEE International Solid-State Circuits Conference. Digest*, pages 270–271, 1997.

[SSN+90] A. Sekiyama, T. Seki, S. Nagai, A. Iwase, N. Suzuki, and M. Hayasaka. A 1 V operating 256-Kbit full CMOS SRAM. *1990 Symposium on VLSI Circuits. Digest of Technical Papers*, pages 53–4, 1990.

[Toh] *http://www.kameyama.ecei.tohoku.ac.jp/index.html*.

[Was] *http://www.ohtsuki.comm.waseda.ac.jp/index-e.html*.

[Wat98] T. Watanabe. Impact of DRAM-Logic Integration on System Performance Chip Architecture, and Design Methodolog y. *ASP-DAC'98 Tutorial 1: Merging DRAM and Logic Possibilities and Challenges*, pages 30–62, 1998.

[WC95] C.-L. Wang and K.-M. Chen. A new VLSI architecture for the full-search vector quantization. *Proc. SPIE - Int. Soc. Opt. Eng. (USA)*, 2501(pt.1):489–98, 1995.

[WE85] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, N.Y., 1985.

[WE93] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design, Second Edition*. Addison-Wesley, N.Y., 1993.

[WFY+97] T. Watanabe, R. Fujita, K. Yanagisawa, H. Tanaka, K. Ayukawa, M. Soga, Y. Tanaka, Y. Sugie, and Y. Nakagome. A modular architecture for a 6.4-Gbyte/s, 8-Mb DRAM-integrated media chip. *IEEE J. Solid-State Circuits (USA)*, 32(5):635–41, 1997.

[WS89] J.P. Wade and C.G. Sodini. A ternary content addressable search engine. *IEEE J. Solid-State Circuits (USA)*, 24(4):1003–13, 1989.

[Yas91] H. Yasuura. Massively parallel processing by functional memories. *Jour. of IPSJ*, 32(oo.12):1260–1267, 1991.

[YF77] S.S. Yau and H.S. Fung. Assosiative Processor Architecture—A Survey. *Computing Surveys*, 9(1), 1977.

[YTT88] H. Yasuura, T. Tsujimoto, and K. Tamaru. Parallel exhaustive search for several NP-complete problems using content addressable memory. *Proc. of 1988 IEEE Int. Symp. on Circuits and Systems*, pages 333–336, 1988.

[YWST91] H. Yasuura, A. Watanabe, R. Sadachi, and K. Tamaru. Functional memory type parallel processor FMPP on a CAM and its applications. *Joint Symp. on Parallel Processing '91*, pages 213–220, 1991.

Publication List

Major Publication

1. K. Kobayashi, K. Tamaru, H. Yasuura, and H. Onodera. “A Bit-parallel Block-parallel Functional Memory Type Parallel Processor Architecture.” *IEICE Trans. on Electron*, vo.E76-C(7):1151–1158, 1993.

2. K. Kobayashi, M. Kinoshita, M. Takeuchi, H. Onodera, and K. Tamaru. “A Memory-based Parallel Processor for Vector Quantization: FMPP-VQ.” *IEICE Trans. on Electron*, E80-C(7):970–975, 1997.

3. K. Kobayashi, N. Nakamura, K. Terada, H. Onodera, and K. Tamaru. “An LSI for Low Bit-Rate Image Compression Using Vector Quantization.” *IEICE Trans. on Electron*, E81-C(5): pages 718–724, 1998.

4. K. Kobayashi, K. Terada, H. Onodera, and K. Tamaru. “A Real-Time Low-Rate Video Compression Algorithm Using Multi-Stage Hierachical Vector Quantization.” *IEICE Trans. on Electron*, E82-A(2): (Under submission), 1999.

Co-authored Publication

1. K. Tamaru, K. Kobayashi and H. Onodera. “Memory based architecture and its implementation scheme named bit-parallel block-parallel functional memory type parallel processor BPBP FMPP.” *Computers & Electrical Engineering*, 24: pages 17–31, 1998.

Conference Presentation

1. K. Kobayashi, H. Onodera, and K. Tamaru. “A bit-parallel block-parallel functional memory type parallel processor LSI for fast addition and Multiplication.” *Symposium on VLSI circuits*, pages 61–62, 1995.

2. K. Kobayashi, M. Kinoshita, M. Takeuchi, H. Onodera, and K. Tamaru. “A memory-based parallel processor for vector quantization.” *22nd European Solid-State Circuits Conference*,

pages 184–187, 1996.

3. K. Kobayashi, M. Kinoshita, M. Takeuchi, H. Onodera and K. Tamaru. “A Functional Memory Type Parallel Processor for Vector Quantization.” *Proc. of the ASP-DAC’97; Asia and South Pacific Design Automation Conference 1997*, pages.665-666, 1997.

4. K. Terada, M. Takeuchi, K. Kobayashi, H. Onodera and K. Tamaru. “Real Time Low Bit-Rate Video Coding Algorithm Using Multi-Stage Hierarchical Vector Quantization.” *Proc. of the ICASSP; International Conference on Accoustics, Speech and Signal Processing*, 2673-2676, 1998.

日本語による口頭発表: Oral Presentations in Japanese.

1. 小林和淑, 田丸啓吉, 安浦寛人：“新しい機能メモリの提案とその応用について”, 平成 3 年電気学会電子・情報・システム部門全国大会講演論文集, 論文 No.C-2-4, pp.209-212 (1991).

2. 小林和淑, 安浦寛人, 田丸啓吉：“ビット並列ブロック並列方式による機能メモリ型並列プロセッサアーキテクチャーの提案”, 情報処理学会第 43 回（平成 3 年後期）全国大会講演論文集, 論文 No.3Q-6, pp.6-57 〜 6-58 (1991).

3. 小林和淑, 小野寺秀俊, 田丸啓吉, 安浦寛人：“ビット並列ブロック並列方式による機能メモリ型並列プロセッサＦＭＰＰの設計－レイアウト面積および動作速度評価－”, 1993 年電子情報通信学会春季大会講演論文集, 論文 No.C-594, p.5-224 (1993).

4. 小林和淑, 小野寺秀俊, 田丸啓吉：“FMPP におけるパストランジスタを用いた並列演算手法”, 1993 年電子情報通信学会秋季大会講演論文集, 論文 No.C-431,p.5-141 (1993).

5. 小林和淑, 竹村秀城, W. Jungsuwadee, 小野寺秀俊, 田丸啓吉：“ビット並列ブロック並列方式による機能メモリ型並列プロセッサの設計”, 電子情報通信学会技術研究報告, SDM93-145, ICD93-139 Nov., pp.37-44 (1993).

6. 竹村秀城, 小林和淑, 小野寺秀俊, 田丸啓吉：“機能メモリ型並列プロセッサ上での離散余弦変換の実現”, 1994 年電子情報通信学会春季大会講演論文集, 論文 No.C-639, p.5-207 (1994).

7. 小林和淑, 小野寺秀俊, 田丸啓吉：“ビット並列ブロック並列型 FMPP アーキテクチャをとるプロトタイプ LSI チップの概要”, 1994 年電子情報通信学会秋季大会講演論文集, 論文 No.C-488, p.166 (1994).

8. W. Jungsuwadee, 小林和淑, 小野寺秀俊, 田丸啓吉：“ビット並列ブロック並列型 FMPP における機能メモリのテスト方法”, 1995 年電子情報通信学会総合大会講演論文集, 論文 No.C-571, p.164 (1995).

9. 安慶武志, 小林和淑, 小野寺秀俊, 田丸啓吉： “BPBP 型 FMPP を用いたプロセッサボードの設計”, 1995 年電子情報通信学会総合大会講演論文集, 論文 No.C-595, p.188 (1995).

10. 木下雅善, 中村典嗣, 小林和淑, 田丸啓吉： “ベクトル量子化に適した機能メモリ型並列プロセッサの設計”, 1995 年電子情報通信学会基礎・境界ソサイエティ大会講演論文集, 論文 No.C-475, p.197 (1995).

11. 清水友人, 小林和淑, 田丸啓吉： “機能メモリ型並列プロセッサを用いたベクトル量子化による画像情報の圧縮”, 1995 年電子情報通信学会基礎・境界ソサイエティ大会講演論文集, 論文 No.C-476, p.198 (1995).

12. 山岡雅直, 小林和淑, 田丸啓吉： “ビット並列ブロック並列方式による機能メモリ型並列プロセッサの設計”, 1996 年電子情報通信学会総合大会講演論文集, 論文 No.C-543, p.159 (1996).

13. 高峰　信, 小林和淑, 田丸啓吉： “加算機能付き画像メモリの設計”, 1996 年電子情報通信学会総合大会講演論文集, 論文 No.C-599, p.215 (1996).

14. 武内昌弘, 木下雅善, 清水友人, 小林和淑, 田丸啓吉： “機能メモリ型並列プロセッサを用いた動画像のベクトル量子化”, 1996 年電子情報通信学会総合大会講演論文集, 論文 No.D-234, p.22 (1996).

15. 小林和淑, 木下雅善, 清水友人, 武内昌弘, 小野寺秀俊, 田丸啓吉： “ベクトル量子化用機能メモリ型並列プロセッサ FMPP-VQ の設計”, 第 9 回回路とシステム軽井沢ワークショップ論文集, PP.353-358 (1996).

16. 小林和淑, 中村典嗣, 武内昌弘, 田丸啓吉: ” ベクトル量子化用メモリーベースプロセッサとその動画像圧縮への応用 ", 1997 年電子情報通信学会総合大会講演論文集, 論文 No.C-12-31, p.167 (1997).

17. 小林幸史, 小林和淑, 田丸啓吉： “加算機能付きメモリの設計”, 1997 年電子情報通信学会エレクトロニクスソサイエティ大会講演論文集, 論文 No.SC-10-6, p.196 (1997).

18. 寺田一彦, 武内昌弘, 中村典嗣, 小林和淑, 田丸啓吉： “ベクトル量子化用機能メモリ型並列プロセッサによる動画像の低ビットレート圧縮システム”, 1997 年電子情報通信学会情報・システムソサイエティ大会講演論文集, 論文 No.D-11-48, p.140 (1997).

19. 武内昌弘, 寺田一彦, 中村典嗣, 小林和淑, 小野寺秀俊, 田丸啓吉： “ベクトル量子化用機能メモリ型並列プロセッサ FMPP-VQ による動画像の低ビットレート圧縮アルゴリズムの提案 ", 第 10 回回路とシステム軽井沢ワークショップ論文集, Apr.21-22, pp.291-296 (1997).

20. 小林和淑, 中村典嗣, 山岡雅直, 小野寺秀俊, 田丸啓吉： “ベクトル量子化用機能メモリ型並列プロセッサ FMPP-VQ64 の設計”, 情報処理学会 DA シンポジウム’97 論文集, pp.13-18 (1997).

21. 小林和淑, 武内昌弘, 寺田和彦, 小野寺秀俊, 田丸啓吉： “ベクトル量子化を用いた低ビットレート動画像圧縮システム”, 第 1 回システム LSI 琵琶湖ワークショップ資料集, pp.365-370, Nov. (1997).

22. 山岡雅直 小林幸史 小林和淑 田丸啓吉, “DRAM を用いた加算機能メモリの設計”, 信学技報, Vol. 97, No. 344, VLD 97-95, pp. 125-132 (1997)

23. 武内昌弘, 寺田一彦, 小林和淑, 田丸啓吉： “ベクトル量子化による低ビットレート動画像圧縮に適した低電力メモリベースプロセッサの設計 ", 1998 年電子情報通信学会総合大会講演論文集, 論文 No. C-12-14, p. 142 (1998).

24. 山岡雅直, 小林幸史, 小林和淑, 田丸啓吉： “DRAM を用いた加算機能メモリ” 1998 年電子情報通信学会総合大会講演論文集, 論文 No. C-12-85, p. 213 (1998)

25. 小林幸史, 山岡雅直, 渡辺航也, 小林和淑, 田丸啓吉： “DRAM を用いた加算機能メモリ” 信学技報, Vol. 97, No. 344, ICD 97-95, pp. 19–26 (1998)

26. 寺田 一彦, 武内 昌弘, 小林 和淑, 田丸 啓吉： “機能メモリ型並列プロセッサによる階層型ベクトル量子化を用いた低ビットレート動画像圧縮システム” 第 11 回回路とシステム軽井沢ワークショップ論文集, Apr.20-21, pp.445–450 (1998).

Acknowledgment

I would like to express my sincere gratitude to Professor Keikichi Tamaru of Kyoto University for his continuous guidance and helpful discussions through this work.

I would like to give great thanks to Professor Hidetoshi Onodera of Kyoto University for his intelligent advice and management for this research.

I would like to express my appreciation to Professor Hiroto Yasuura of Kyushu University. Prof. Tamaru and Prof. Yasuura started this work.

I would like to thank all the members of Tamaru Laboratory who have contributed this work. The list of the members is as follows: A. Nakano, T. Tujimoto, A. Watanabe, R. Sadachi, K.M. Lu, H. Takemura, J. Wasarin, K. Kamei, M. Kinoshita, T. Ankei, T. Shimizu, H. Shimizu, N. Nakamura, S. Takamine, M. Takeuchi, M. Yamaoka, Y. Kobayashi, K. Terada, C. Tan.

I am also deeply indebted to my wife Etsuko for her every-day's house work. I would like to express my appreciation to my father Seiichi and mother Michiko. Finally, may God bless my dear daughter, Momoka.